

# **Documentação do PostgreSQL 8.0.0**

**Projeto de Tradução para o Português do Brasil**  
(<http://sourceforge.net/projects/pgdocptbr/>)

**The PostgreSQL Global Development Group**

**Rio de Janeiro, 29 de dezembro de 2005**

## **Documentação do PostgreSQL 8.0.0**

Projeto de Tradução para o Português do Brasil (<http://sourceforge.net/projects/pgdocptbr/>)

The PostgreSQL Global Development Group

Copyright © 1996-2005 por The PostgreSQL Global Development Group

### **Legal Notice**

PostgreSQL is Copyright © 1996-2005 by the PostgreSQL Global Development Group and is distributed under the terms of the license of the University of California below.

Postgres95 is Copyright © 1994-5 by the Regents of the University of California.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN “AS-IS” BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

### **Traduzido por**

- Halley Pacheco de Oliveira (<[halleypo@users.sourceforge.net](mailto:halleypo@users.sourceforge.net)>)

### **Revisado por**

- Diogo de Oliveira Biazus
- Marcia Tonon

# Sumário

Prefácio.....	i
1. O que é o PostgreSQL? .....	i
2. Uma breve história do PostgreSQL .....	i
3. Convenções .....	iii
4. Outras informações .....	iii
5. Guia para informar erros .....	iv
I. Tutorial.....	8
Capítulo 1. Início .....	1
1.1. Instalação .....	1
1.2. Fundamentos da arquitetura .....	1
1.3. Criação de banco de dados .....	1
1.4. Acesso a banco de dados .....	3
Capítulo 2. A linguagem SQL .....	6
2.1. Introdução .....	6
2.2. Conceitos .....	6
2.3. Criação de tabelas .....	6
2.4. Inserção de linhas em tabelas .....	7
2.5. Consultar tabelas .....	8
2.6. Junções entre tabelas .....	9
2.7. Funções de agregação.....	11
2.8. Atualizações .....	12
2.9. Exclusões .....	13
Capítulo 3. Funcionalidades avançadas.....	14
3.1. Introdução .....	14
3.2. Visões.....	14
3.3. Chaves estrangeiras .....	14
3.4. Transações.....	15
3.5. Herança .....	16
3.6. Conclusão .....	18
II. A linguagem SQL.....	19
Capítulo 4. Sintaxe da linguagem SQL .....	20
4.1. Estrutura léxica.....	20
4.2. Expressões de valor.....	29
Capítulo 5. Definição de dados .....	37
5.1. Noções básicas de tabela .....	37
5.2. Valor padrão.....	38
5.3. Restrições .....	39
5.4. Colunas do sistema.....	48
5.5. Herança .....	49
5.6. Modificação de tabelas.....	51
5.7. Privilégios .....	53
5.8. Esquemas .....	54
5.9. Outros objetos de banco de dados .....	59
5.10. Acompanhando as dependências.....	59
Capítulo 6. Manipulação de dados .....	61
6.1. Inserção de dados.....	61
6.2. Atualização de dados .....	62
6.3. Exclusão de dados .....	62
Capítulo 7. Consultas .....	64
7.1. Visão geral .....	64
7.2. Expressões de tabela.....	64
7.3. Listas de seleção.....	75
7.4. Combinação de consultas .....	76
7.5. Ordenação de linhas .....	77
7.6. LIMIT e OFFSET .....	78
Capítulo 8. Tipos de dado.....	80
8.1. Tipos numéricos .....	81
8.2. Tipos monetários.....	85

8.3. Tipos para cadeias de caracteres .....	86
8.4. Tipos de dado binários .....	89
8.5. Tipos para data e hora .....	91
8.6. Tipo booleano.....	97
8.7. Tipos geométricos.....	98
8.8. Tipos para endereço de rede.....	100
8.9. Tipos para cadeias de bits .....	102
8.10. Matrizes.....	102
8.11. Tipos compostos.....	111
8.12. Tipos identificadores de objeto .....	114
8.13. Pseudotipos.....	116
Capítulo 9. Funções e Operadores .....	120
9.1. Operadores lógicos .....	120
9.2. Operadores de comparação .....	120
9.3. Funções e operadores matemáticos.....	122
9.4. Funções e operadores para cadeias de caracteres.....	126
9.5. Funções e operadores para cadeias binárias .....	139
9.6. Funções e operadores para cadeias de bits .....	140
9.7. Correspondência com padrão.....	141
9.8. Funções para formatar tipo de dado.....	163
9.9. Funções e operadores para data e hora.....	169
9.10. Funções e operadores geométricos.....	186
9.11. Funções e operadores para endereço de rede.....	189
9.12. Funções para manipulação de seqüências.....	190
9.13. Expressões condicionais.....	192
9.14. Funções e operadores para matrizes .....	194
9.15. Funções de agregação.....	195
9.16. Expressões de subconsulta .....	197
9.17. Comparações de linha e de matriz .....	201
9.18. Funções que retornam conjunto .....	202
9.19. Funções de informação do sistema .....	203
9.20. Funções para administração do sistema .....	212
Capítulo 10. Conversão de tipo .....	215
10.1. Visão geral .....	215
10.2. Operadores .....	216
10.3. Funções.....	218
10.4. Armazenamento de valor .....	220
10.5. Construções UNION, CASE e ARRAY .....	221
Capítulo 11. Índices.....	223
11.1. Introdução .....	223
11.2. Tipos de índice .....	224
11.3. Índices com várias colunas .....	225
11.4. Índices únicos.....	226
11.5. Índices em expressões.....	226
11.6. Classes de operadores .....	227
11.7. Índices parciais.....	228
11.8. Examinar a utilização do índice .....	229
Capítulo 12. Controle de simultaneidade .....	232
12.1. Introdução .....	232
12.2. Isolamento da transação.....	232
12.3. Bloqueio explícito .....	235
12.4. Verificação da consistência dos dados no nível do aplicativo.....	238
12.5. Bloqueio e índices.....	238
Capítulo 13. Dicas de desempenho .....	240
13.1. Utilização do comando EXPLAIN.....	240
13.2. Estatísticas utilizadas pelo planejador .....	243
13.3. Controle do planejador com cláusulas JOIN explícitas.....	244
13.4. Carga dos dados no banco .....	246
III. Administração do servidor .....	248
Capítulo 14. Instruções de instalação .....	249
14.1. Resumo da instalação.....	249
14.2. Requisitos.....	249
14.3. Obtenção dos arquivos fonte .....	251

14.4. Se estiver atualizando .....	251
14.5. Procedimento de instalação .....	252
14.6. Configurações de pós-instalação .....	257
14.7. Plataformas suportadas .....	258
14.8. Instalação no Fedora Core 3 .....	261
Capítulo 15. Instalação apenas do cliente no Windows .....	265
Capítulo 16. Ambiente do servidor em tempo de execução .....	266
16.1. A conta de usuário do PostgreSQL .....	266
16.2. Criação do agrupamento de bancos de dados .....	266
16.3. Inicialização do servidor de banco de dados .....	267
16.4. Configuração em tempo de execução .....	270
16.5. Gerência dos recursos do núcleo .....	292
16.6. Parada do servidor .....	298
16.7. Conexões TCP/IP seguras com SSL .....	298
16.8. Conexões TCP/IP seguras por túneis SSH .....	299
Capítulo 17. Usuários do banco de dados e privilégios .....	302
17.1. Usuários de banco de dados .....	302
17.2. Atributos do usuário .....	303
17.3. Grupos .....	303
17.4. Privilégios .....	304
17.5. Funções e gatilhos .....	304
Capítulo 18. Gerenciamento de bancos de dados .....	305
18.1. Visão geral .....	305
18.2. Criação de banco de dados .....	305
18.3. Bancos de dado modelo .....	306
18.4. Configuração do banco de dados .....	308
18.5. Remoção do banco de dados .....	308
18.6. Espaços de tabelas .....	308
Capítulo 19. Autenticação de clientes .....	312
19.1. O arquivo pg_hba.conf .....	312
19.2. Métodos de autenticação .....	316
19.3. Problemas de autenticação .....	319
Capítulo 20. Idioma .....	321
20.1. Suporte a idioma .....	321
20.2. Suporte a conjuntos de caracteres .....	323
Capítulo 21. Rotinas de manutenção do banco de dados .....	328
21.1. Rotina de Limpeza .....	328
21.2. Rotina de reindexação .....	331
21.3. Manutenção do arquivo de registro .....	331
Capítulo 22. Criação e restauração de cópias de segurança .....	333
22.1. Método SQL-dump .....	333
22.2. Cópia de segurança no nível de sistema de arquivo .....	335
22.3. Cópia de segurança em-linha .....	336
22.4. Migração entre versões .....	342
Capítulo 23. Monitoramento das atividades do banco de dados .....	344
23.1. Ferramentas padrão do Unix .....	344
23.2. O coletor de estatísticas .....	344
23.3. Ver os bloqueios .....	348
Capítulo 24. Monitoramento da utilização de disco .....	350
24.1. Determinação da utilização de disco .....	350
24.2. Falha de disco cheio .....	352
Capítulo 25. Registro prévio da escrita (WAL) .....	353
25.1. Benefícios do WAL .....	353
25.2. Configuração do WAL .....	353
25.3. Internamente .....	355
Capítulo 26. Testes de regressão .....	356
26.1. Execução dos testes .....	356
26.2. Avaliação dos testes .....	357
26.3. Arquivos de comparação específicos de plataformas .....	359
IV. Interfaces cliente .....	360
Capítulo 27. libpq - Biblioteca C .....	361
27.1. Funções de controle da conexão com o banco de dados .....	361
27.2. Funções de status da conexão .....	366

27.3. Funções de execução de comando .....	368
27.4. Processamento de comandos assíncronos .....	378
27.5. Cancelamento de comandos em andamento .....	381
27.6. A interface de caminho-rápido .....	382
27.7. Notificação assíncrona .....	382
27.8. Funções associadas ao comando COPY .....	383
27.9. Funções de controle .....	387
27.10. Processamento de notas .....	387
27.11. Variáveis de Ambiente .....	388
27.12. O arquivo de senhas .....	389
27.13. Suporte a SSL .....	390
27.14. Comportamento dos programas com fluxo de execução .....	390
27.15. Construção de programas que utilizam a libpq .....	390
27.16. Programas exemplo .....	391
Capítulo 28. Objetos grandes .....	400
28.1. Histórico .....	400
28.2. Funcionalidades da implementação .....	400
28.3. Interfaces cliente .....	400
28.4. Funções do lado servidor .....	402
28.5. Programa exemplo .....	403
Capítulo 29. ECPG - SQL incorporado à linguagem C .....	408
29.1. O conceito .....	408
29.2. Conexão com o servidor de banco de dados .....	408
29.3. Fechamento de conexão .....	409
29.4. Execução de comandos SQL .....	409
29.5. Escolha da conexão .....	410
29.6. Utilização de variáveis hospedeiras .....	411
29.7. SQL dinâmico .....	413
29.8. Utilização das áreas descritoras de SQL .....	414
29.9. Tratamento de erro .....	415
29.10. Inclusão de arquivos .....	419
29.11. Processamento dos programas com SQL incorporado .....	420
29.12. Funções da biblioteca .....	420
29.13. Internamente .....	421
29.14. Exemplos .....	422
Capítulo 30. O esquema de informações .....	432
30.1. O esquema .....	432
30.2. Tipos de dado .....	432
30.3. information_schema_catalog_name .....	433
30.4. applicable_roles .....	433
30.5. check_constraints .....	434
30.6. column_domain_usage .....	434
30.7. column_privileges .....	435
30.8. column_udt_usage .....	436
30.9. columns .....	436
30.10. constraint_column_usage .....	439
30.11. constraint_table_usage .....	440
30.12. data_type_privileges .....	441
30.13. domain_constraints .....	441
30.14. domain_udt_usage .....	442
30.15. domains .....	442
30.16. element_types .....	446
30.17. enabled_roles .....	448
30.18. key_column_usage .....	448
30.19. parameters .....	449
30.20. referential_constraints .....	451
30.21. role_column_grants .....	452
30.22. role_routine_grants .....	452
30.23. role_table_grants .....	453
30.24. role_usage_grants .....	453
30.25. routine_privileges .....	454
30.26. routines .....	454
30.27. schemata .....	457

30.28. sql_features.....	458
30.29. sql_implementation_info.....	458
30.30. sql_languages.....	458
30.31. sql_packages.....	459
30.32. sql_sizing.....	460
30.33. sql_sizing_profiles.....	461
30.34. table_constraints.....	461
30.35. table_privileges.....	462
30.36. tables.....	463
30.37. triggers.....	464
30.38. usage_privileges.....	465
30.39. view_column_usage.....	465
30.40. view_table_usage.....	466
30.41. views.....	466
V. Programação servidor.....	474
Capítulo 31. Estendendo a linguagem SQL.....	475
31.1. Como funciona a extensibilidade.....	475
31.2. O sistema de tipos de dado do PostgreSQL.....	475
31.3. Funções definidas pelo usuário.....	476
31.4. Funções na linguagem de comando (SQL).....	477
31.5. Sobrecarga de função.....	483
31.6. Categorias de volatilidade de função.....	484
31.7. Funções nas linguagens procedurais.....	485
31.8. Funções internas.....	485
31.9. Funções na linguagem C.....	486
31.10. Agregações definidas pelo usuário.....	515
31.11. Tipos definidos pelo usuário.....	517
31.12. Operadores definidos pelo usuário.....	519
31.13. Informações de otimização do operador.....	520
31.14. Interfacing Extensions To Indexes.....	524
Capítulo 32. Gatilhos.....	531
32.1. Visão geral do comportamento dos gatilhos.....	531
32.2. Visibilidade das mudanças nos dados.....	532
32.3. Escrita de funções de gatilho em C.....	532
32.4. Um exemplo completo.....	534
Capítulo 33. O sistema de regras.....	538
33.1. A árvore de comando.....	538
33.2. As visões e o sistema de regras.....	539
33.3. Regras para INSERT, UPDATE e DELETE.....	545
33.4. Regras e privilégios.....	554
33.5. Regras e status dos comandos.....	555
33.6. Regras versus gatilhos.....	555
Capítulo 34. Linguagens procedurais.....	558
34.1. Instalação de linguagem procedural.....	558
Capítulo 35. PL/pgSQL - Linguagem procedural SQL.....	560
35.1. Visão geral.....	560
35.2. Dicas para desenvolvimento em PL/pgSQL.....	561
35.3. Estrutura da linguagem PL/pgSQL.....	563
35.4. Declarações.....	564
35.5. Expressões.....	567
35.6. Instruções básicas.....	568
35.7. Estruturas de controle.....	572
35.8. Cursores.....	578
35.9. Erros e mensagens.....	581
35.10. Procedimentos de gatilho.....	582
35.11. Conversão do PL/SQL do Oracle para o PL/pgSQL do PostgreSQL.....	591
Capítulo 36. PL/Tcl - Tcl Procedural Language.....	600
36.1. Visão geral.....	600
36.2. PL/Tcl Functions and Arguments.....	600
36.3. Data Values in PL/Tcl.....	601
36.4. Global Data in PL/Tcl.....	601
36.5. Database Access from PL/Tcl.....	601
36.6. Trigger Procedures in PL/Tcl.....	603

36.7. Modules and the unknown command .....	605
36.8. Tcl Procedure Names.....	605
Capítulo 37. PL/Perl - Perl Procedural Language .....	606
37.1. PL/Perl Functions and Arguments .....	606
37.2. Database Access from PL/Perl .....	607
37.3. Data Values in PL/Perl .....	609
37.4. Global Values in PL/Perl .....	609
37.5. Trusted and Untrusted PL/Perl.....	609
37.6. PL/Perl Triggers .....	610
37.7. Limitations and Missing Features.....	611
Capítulo 38. PL/Python - Python Procedural Language.....	612
38.1. PL/Python Functions .....	612
38.2. Trigger Functions .....	612
38.3. Database Access .....	613
Capítulo 39. Server Programming Interface.....	614
39.1. Interface Functions.....	614
SPI_connect .....	615
SPI_finish .....	616
SPI_push .....	617
SPI_pop.....	618
SPI_execute .....	619
SPI_exec .....	622
SPI_prepare .....	623
SPI_getargcount.....	624
SPI_getargtypeid.....	625
SPI_is_cursor_plan .....	626
SPI_execute_plan .....	627
SPI_execp .....	628
SPI_cursor_open.....	629
SPI_cursor_find.....	630
SPI_cursor_fetch.....	631
SPI_cursor_move.....	632
SPI_cursor_close .....	633
SPI_saveplan .....	634
39.2. Interface Support Functions .....	634
SPI_fname.....	635
SPI_fnumber .....	636
SPI_getvalue .....	637
SPI_getbinval .....	638
SPI_gettype .....	639
SPI_gettypeid .....	640
SPI_getrelname.....	641
39.3. Memory Management .....	641
SPI_palloc .....	642
SPI_repalloc .....	643
SPI_pfree.....	644
SPI_copytuple .....	645
SPI_returntuple .....	646
SPI_modifytuple .....	647
SPI_freetuple.....	648
SPI_freetuptable.....	649
SPI_freeplan.....	650
39.4. Visibility of Data Changes .....	650
39.5. Exemplos.....	650
VI. Referência .....	654
I. Comandos SQL .....	655
ABORT .....	656
ALTER AGGREGATE .....	657
ALTER CONVERSION.....	658
ALTER DATABASE.....	659
ALTER DOMAIN .....	661
ALTER FUNCTION .....	663
ALTER GROUP.....	664



ALTER INDEX.....	665
ALTER LANGUAGE.....	667
ALTER OPERATOR.....	668
ALTER OPERATOR CLASS.....	669
ALTER SCHEMA.....	670
ALTER SEQUENCE.....	671
ALTER TABLE.....	673
ALTER TABLESPACE.....	678
ALTER TRIGGER.....	679
ALTER TYPE.....	680
ALTER USER.....	681
ANALYZE.....	684
BEGIN.....	686
CHECKPOINT.....	688
CLOSE.....	689
CLUSTER.....	690
COMMENT.....	692
COMMIT.....	695
COPY.....	696
CREATE AGGREGATE.....	703
CREATE CAST.....	706
CREATE CONSTRAINT TRIGGER.....	709
CREATE CONVERSION.....	710
CREATE DATABASE.....	712
CREATE DOMAIN.....	714
CREATE FUNCTION.....	717
CREATE GROUP.....	722
CREATE INDEX.....	723
CREATE LANGUAGE.....	726
CREATE OPERATOR.....	728
CREATE OPERATOR CLASS.....	731
CREATE RULE.....	734
CREATE SCHEMA.....	736
CREATE SEQUENCE.....	738
CREATE TABLE.....	741
CREATE TABLE AS.....	750
CREATE TABLESPACE.....	752
CREATE TRIGGER.....	753
CREATE TYPE.....	756
CREATE USER.....	761
CREATE VIEW.....	763
DEALLOCATE.....	765
DECLARE.....	766
DELETE.....	769
DROP AGGREGATE.....	771
DROP CAST.....	772
DROP CONVERSION.....	773
DROP DATABASE.....	774
DROP DOMAIN.....	775
DROP FUNCTION.....	776
DROP GROUP.....	777
DROP INDEX.....	778
DROP LANGUAGE.....	779
DROP OPERATOR.....	780
DROP OPERATOR CLASS.....	781
DROP RULE.....	782
DROP SCHEMA.....	783
DROP SEQUENCE.....	784
DROP TABLE.....	785
DROP TABLESPACE.....	786
DROP TRIGGER.....	787
DROP TYPE.....	788
DROP USER.....	789

DROP VIEW .....	790
END .....	791
EXECUTE.....	792
EXPLAIN .....	793
FETCH.....	796
GRANT .....	800
INSERT .....	804
LISTEN.....	806
LOAD .....	807
LOCK.....	808
MOVE .....	810
NOTIFY .....	811
PREPARE .....	813
REINDEX.....	815
RELEASE SAVEPOINT .....	817
RESET.....	818
REVOKE.....	819
ROLLBACK .....	821
ROLLBACK TO SAVEPOINT .....	822
SAVEPOINT .....	824
SELECT.....	826
SELECT INTO.....	836
SET.....	838
SET CONSTRAINTS.....	841
SET SESSION AUTHORIZATION .....	842
SET TRANSACTION.....	843
SHOW .....	845
START TRANSACTION.....	847
TRUNCATE .....	848
UNLISTEN.....	849
UPDATE .....	850
VACUUM.....	852
II. Aplicativos cliente do PostgreSQL .....	854
clusterdb .....	855
createdb.....	857
createlang.....	859
createuser.....	861
dropdb .....	864
droplang.....	866
dropuser .....	868
ecpg.....	870
pg_config.....	872
pg_dump.....	874
pg_dumpall.....	880
pg_restore .....	884
psql .....	889
vacuumdb.....	912
III. Aplicativos do servidor PostgreSQL .....	915
initdb .....	916
ipcclean .....	919
pg_controldata.....	920
pg_ctl.....	921
pg_resetxlog.....	925
postgres.....	927
postmaster.....	930
VII. Internamente .....	934
Capítulo 40. Visão geral da estrutura interna do PostgreSQL .....	935
40.1. O caminho do comando .....	935
40.2. Como as conexões são estabelecidas.....	935
40.3. O estágio de análise.....	936
40.4. O sistema de regras do PostgreSQL .....	937
40.5. Planejador/Otimizador.....	937
40.6. Executor .....	938

Capítulo 41. Catálogos do sistema.....	940
41.1. Visão geral .....	940
41.2. pg_aggregate .....	941
41.3. pg_am .....	941
41.4. pg_amop .....	942
41.5. pg_amproc .....	943
41.6. pg_attrdef .....	943
41.7. pg_attribute .....	943
41.8. pg_cast.....	944
41.9. pg_class .....	945
41.10. pg_constraint.....	947
41.11. pg_conversion .....	948
41.12. pg_database.....	948
41.13. pg_depend .....	949
41.14. pg_description .....	950
41.15. pg_group .....	950
41.16. pg_index.....	950
41.17. pg_inherits.....	951
41.18. pg_language.....	952
41.19. pg_largeobject.....	952
41.20. pg_listener.....	952
41.21. pg_namespace.....	953
41.22. pg_opclass .....	953
41.23. pg_operator .....	954
41.24. pg_proc .....	954
41.25. pg_rewrite.....	956
41.26. pg_shadow .....	956
41.27. pg_statistic .....	957
41.28. pg_tablespace.....	958
41.29. pg_trigger .....	958
41.30. pg_type.....	959
41.31. Visões do sistema .....	961
41.32. pg_indexes .....	962
41.33. pg_locks .....	962
41.34. pg_rules.....	963
41.35. pg_settings .....	963
41.36. pg_stats.....	964
41.37. pg_tables.....	965
41.38. pg_user .....	966
41.39. pg_views .....	966
Capítulo 42. Frontend/Backend Protocol.....	967
42.1. Visão geral .....	967
42.2. Message Flow .....	968
42.3. Message Data Types .....	976
42.4. Message Formats.....	976
42.5. Error and Notice Message Fields.....	988
42.6. Summary of Changes since Protocol 2.0.....	989
Capítulo 43. Convenções de codificação do PostgreSQL .....	991
43.1. Formatação .....	991
43.2. Mensagens de erro geradas pelo servidor.....	991
43.3. Guia de estilo para mensagens de erro .....	993
Capítulo 44. Suporte a idioma nativo .....	998
44.1. Para o tradutor .....	998
44.2. Para o programador .....	1001
Capítulo 45. Writing A Procedural Language Handler .....	1004
Capítulo 46. Genetic Query Optimizer.....	1006
46.1. Query Handling as a Complex Optimization Problem.....	1006
46.2. Genetic Algorithms .....	1006
46.3. Genetic Query Optimization (GEQO) in PostgreSQL .....	1007
46.4. Further Reading .....	1008
Capítulo 47. Index Cost Estimation Functions.....	1009
Capítulo 48. GiST Indexes .....	1011
48.1. Introdução .....	1011

48.2. Extensibility .....	1011
48.3. Implementation .....	1011
48.4. Limitations .....	1012
48.5. Exemplos .....	1012
Capítulo 49. Armazenamento físico dos bancos de dados .....	1013
49.1. Organização dos arquivos de banco de dados .....	1013
49.2. TOAST .....	1014
49.3. Disposição das páginas de banco de dados .....	1015
Capítulo 50. BKI Backend Interface .....	1020
50.1. BKI File Format .....	1020
50.2. BKI Commands .....	1020
50.3. Example .....	1021
VIII. Apêndices .....	1022
Apêndice A. Códigos de erro do PostgreSQL .....	1023
Apêndice B. Apoio a data e hora .....	1030
B.1. Interpretação de data e hora .....	1030
B.2. Palavras chave para data e hora .....	1031
B.3. História das unidades .....	1047
Apêndice C. Palavras chave do SQL .....	1049
Apêndice D. Conformidade com o padrão SQL .....	1073
D.1. Funcionalidades suportadas .....	1074
D.2. Funcionalidades não suportadas .....	1082
Apêndice E. Release Notes .....	1089
E.1. Release 8.0 .....	1089
E.2. Release 7.4.6 .....	1100
E.3. Release 7.4.5 .....	1101
E.4. Release 7.4.4 .....	1101
E.5. Release 7.4.3 .....	1102
E.6. Release 7.4.2 .....	1102
E.7. Release 7.4.1 .....	1104
E.8. Release 7.4 .....	1105
E.9. Release 7.3.8 .....	1117
E.10. Release 7.3.7 .....	1117
E.11. Release 7.3.6 .....	1118
E.12. Release 7.3.5 .....	1118
E.13. Release 7.3.4 .....	1119
E.14. Release 7.3.3 .....	1119
E.15. Release 7.3.2 .....	1121
E.16. Release 7.3.1 .....	1122
E.17. Release 7.3 .....	1123
E.18. Release 7.2.6 .....	1132
E.19. Release 7.2.5 .....	1133
E.20. Release 7.2.4 .....	1133
E.21. Release 7.2.3 .....	1134
E.22. Release 7.2.2 .....	1134
E.23. Release 7.2.1 .....	1135
E.24. Release 7.2 .....	1135
E.25. Release 7.1.3 .....	1143
E.26. Release 7.1.2 .....	1144
E.27. Release 7.1.1 .....	1144
E.28. Release 7.1 .....	1144
E.29. Release 7.0.3 .....	1148
E.30. Release 7.0.2 .....	1149
E.31. Release 7.0.1 .....	1149
E.32. Release 7.0 .....	1150
E.33. Release 6.5.3 .....	1156
E.34. Release 6.5.2 .....	1156
E.35. Release 6.5.1 .....	1157
E.36. Release 6.5 .....	1158
E.37. Release 6.4.2 .....	1162
E.38. Release 6.4.1 .....	1162
E.39. Release 6.4 .....	1163
E.40. Release 6.3.2 .....	1167

E.41. Release 6.3.1 .....	1167
E.42. Release 6.3 .....	1168
E.43. Release 6.2.1 .....	1172
E.44. Release 6.2 .....	1172
E.45. Release 6.1.1 .....	1174
E.46. Release 6.1 .....	1175
E.47. Release 6.0 .....	1177
E.48. Release 1.09 .....	1179
E.49. Release 1.02 .....	1179
E.50. Release 1.01 .....	1180
E.51. Release 1.0 .....	1182
E.52. Postgres95 Release 0.03 .....	1183
E.53. Postgres95 Release 0.02 .....	1185
E.54. Postgres95 Release 0.01 .....	1185
Apêndice F. O repositório CVS .....	1187
F.1. Obtenção do código fonte via CVS anônimo .....	1187
F.2. Organização da árvore do CVS .....	1188
F.3. Obtenção do código fonte via CVSup .....	1189
Apêndice G. Documentação .....	1194
G.1. DocBook .....	1194
G.2. Conjunto de ferramentas .....	1194
G.3. Geração da documentação .....	1197
G.4. Criação da documentação .....	1200
G.5. Guia de estilo .....	1201
Apêndice H. Projetos externos .....	1204
H.1. Interfaces desenvolvidas externamente .....	1204
H.2. Extensões .....	1205



# Prefácio

Este livro é a documentação oficial do PostgreSQL, escrita por seus desenvolvedores e outros voluntários em paralelo ao desenvolvimento do software. Nesta documentação estão descritas todas as funcionalidades suportadas oficialmente pela versão corrente.

Para tornar uma grande quantidade de informações sobre o PostgreSQL gerenciável, este livro está organizado em várias partes. Cada parte se destina a uma classe diferente de usuários, ou a usuários com graus diferentes de experiência com o PostgreSQL:

- Parte I é uma introdução informal para os novos usuários.
- Parte II documenta o ambiente da linguagem de comandos SQL, incluindo tipos de dado e funções, assim como ajuste de desempenho no nível de usuário. Todo usuário do PostgreSQL deve ler esta parte.
- Parte III descreve a instalação e a administração do servidor. Todas as pessoas responsáveis pelo servidor PostgreSQL, seja para uso privativo ou por outros, devem ler esta parte.
- Parte IV descreve as interfaces de programação para os programas cliente do PostgreSQL.
- Parte V contém informações para os usuários avançados sobre a capacidade de extensão do servidor. Os tópicos incluem, por exemplo, tipos de dado definidos pelos usuários e funções.
- Parte VI contém informações de referência sobre os comandos SQL, programas cliente e servidor. Esta parte apóia as outras partes, com informações estruturadas classificadas por comando ou por programa.
- Parte VII contém diversas informações úteis para os desenvolvedores do PostgreSQL.

## 1. O que é o PostgreSQL?

O PostgreSQL é um sistema gerenciador de banco de dados objeto-relacional (SGBDOR), <sup>1 2</sup> baseado no POSTGRES Versão 4.2 (<http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html>) desenvolvido pelo Departamento de Ciência da Computação da Universidade da Califórnia em Berkeley. O POSTGRES foi pioneiro em vários conceitos que somente se tornaram disponíveis muito mais tarde em alguns sistemas de banco de dados comerciais.

O PostgreSQL é um descendente de código fonte aberto deste código original de Berkeley. É suportada grande parte do padrão SQL:2003, além de serem oferecidas muitas funcionalidades modernas, como:

- comandos complexos
- chaves estrangeiras
- gatilhos
- visões
- integridade transacional
- controle de simultaneidade multiversão

Além disso, o PostgreSQL pode ser estendido pelo usuário de muitas maneiras como, por exemplo, adicionando novos

- tipos de dado
- funções
- operadores
- funções de agregação
- métodos de índice
- linguagens procedurais

Devido à sua licença liberal, o PostgreSQL pode ser utilizado, modificado e distribuído por qualquer pessoa para qualquer finalidade, seja privada, comercial ou acadêmica, livre de encargos.

## 2. Uma breve história do PostgreSQL

O sistema gerenciador de banco de dados objeto-relacional hoje conhecido por PostgreSQL, é derivado do pacote POSTGRES escrito na Universidade da Califórnia em Berkeley. Com mais de uma década de desenvolvimento por trás, o PostgreSQL é atualmente o mais avançado banco de dados de código aberto disponível em qualquer lugar.

## 2.1. O projeto POSTGRES de Berkeley

O projeto POSTGRES, liderado pelo Professor Michael Stonebraker, foi patrocinado pela DARPA (Defense Advanced Research Projects Agency), pelo ARO (Army Research Office), pela NSF (National Science Foundation) e pela ESL, Inc. A implementação do POSTGRES começou em 1986. Os conceitos iniciais para o sistema foram apresentados em *The design of POSTGRES*, e a definição do modelo de dados inicial foi descrita em *The POSTGRES data model*. O projeto do sistema de regras desta época foi descrito em *The design of the POSTGRES rules system*. Os fundamentos lógicos e a arquitetura do gerenciador de armazenamento foram detalhados em *The design of the POSTGRES storage system*.

O Postgres passou por várias versões principais desde então. A primeira “versão de demonstração” do sistema se tornou operacional em 1987, e foi exibida em 1988 na Conferência ACM-SIGMOD. A versão 1, descrita em *The implementation of POSTGRES*, foi liberada para alguns poucos usuários externos em junho de 1989. Em resposta à crítica ao primeiro sistema de regras (*A commentary on the POSTGRES rules system*), o sistema de regras foi reprojetoado (*On Rules, Procedures, Caching and Views in Database Systems*), e a versão 2 foi liberada em junho de 1990, contendo um novo sistema de regras. A versão 3 surgiu em 1991 adicionando suporte a múltiplos gerenciadores de armazenamento, um executor de comandos melhorado, e um sistema de regras reescrito. Em sua maior parte as versões seguintes, até o Postgres95 (veja abaixo), focaram a portabilidade e a confiabilidade.

O POSTGRES tem sido usado para implementar muitas aplicações diferentes de pesquisa e de produção, incluindo: sistema de análise de dados financeiros, pacote de monitoração de desempenho de motor a jato, banco de dados de acompanhamento de asteróide, banco de dados de informações médicas, e vários sistemas de informações geográficas. O POSTGRES também tem sido usado como ferramenta educacional por várias universidades. Por fim, a Illustra Information Technologies (posteriormente incorporada pela Informix (<http://www.informix.com/>), que agora pertence à IBM (<http://www.ibm.com/>)) pegou o código e comercializou. O POSTGRES se tornou o gerenciador de dados principal do projeto de computação científica Sequoia 2000 ([http://meteora.ucsd.edu/s2k/s2k\\_home.html](http://meteora.ucsd.edu/s2k/s2k_home.html)) no final de 1992.

O tamanho da comunidade de usuários externos praticamente dobrou durante o ano de 1993. Começou a ficar cada vez mais óbvio que a manutenção do código do protótipo e o suporte estavam consumindo grande parte do tempo que deveria ser dedicado a pesquisas de banco de dados. Em um esforço para reduzir esta sobrecarga de suporte, o projeto do POSTGRES de Berkeley terminou oficialmente na versão 4.2.

## 2.2. O Postgres95

Em 1994, Andrew Yu e Jolly Chen adicionaram um interpretador da linguagem SQL ao POSTGRES. Sob um novo nome, o Postgres95 foi em seguida liberado na Web para encontrar seu próprio caminho no mundo, como descendente de código aberto do código original do POSTGRES de Berkeley.

O código do Postgres95 era totalmente escrito em ANSI C, com tamanho reduzido em 25%. Muitas mudanças internas melhoraram o desempenho e a facilidade de manutenção. O Postgres95 versão 1.0.x era 30-50% mais rápido que o POSTGRES versão 4.2, pelo Wisconsin Benchmark. Além da correção de erros, as principais melhorias foram as seguintes:

- A linguagem de comandos `PostQUEL` foi substituída pela linguagem SQL (implementada no servidor). Não foram permitidas subconsultas até o PostgreSQL (veja abaixo), mas estas podiam ser simuladas no Postgres95 por meio de funções SQL definidas pelo usuário. As funções de agregação foram reimplementadas. Também foi adicionado suporte a cláusula `GROUP BY` nas consultas.
- Foi fornecido um novo programa para executar comandos SQL interativos, o `psql`, utilizando o Readline do GNU, que substituiu com vantagens o programa `monitor` antigo.
- Uma nova biblioteca cliente, a `libpgtcl`, dava suporte a clientes baseados no Tcl. O interpretador de comandos `pgtclsh` fornecia novos comandos Tcl para interfacear programas Tcl com o servidor Postgres95.
- A interface para objetos grandes foi revisada. A inversão de objetos grandes<sup>3</sup> era o único mecanismo para armazenar objetos grandes (O sistema de arquivos inversão foi removido).
- O sistema de regras no nível de instância foi removido. As regras ainda eram disponíveis como regras de reescrita.
- Um breve tutorial introduzindo as funcionalidades regulares da linguagem SQL, assim como as do Postgres95, foi distribuído junto com o código fonte.



- O utilitário `make` do GNU (em vez do `make` do BSD) foi utilizado para a geração. Além disso, o Postgres95 podia ser compilado com o GCC sem correções (o alinhamento de dados para a precisão dupla foi corrigido).

## 2.3. O PostgreSQL

Em 1996 ficou claro que o nome “Postgres95” não resistiria ao teste do tempo. Foi escolhido um novo nome, PostgreSQL, para refletir o relacionamento entre o POSTGRES original e as versões mais recentes com capacidade SQL. Ao mesmo tempo, foi mudado o número da versão para começar em 6.0, colocando a numeração de volta à sequência original começada pelo projeto POSTGRES de Berkeley.

A ênfase durante o desenvolvimento do Postgres95 era identificar e compreender os problemas existentes no código do servidor. Com o PostgreSQL a ênfase foi reorientada para o aumento das funcionalidades e recursos, embora o trabalho continuasse em todas as áreas.

Os detalhes sobre o que aconteceu com o PostgreSQL desde então podem ser encontrados no Apêndice E.

## 3. Convenções

Este livro utiliza as seguintes convenções tipográficas para marcar certas partes do texto: novos termos, frases estrangeiras e outras passagens importantes são enfatizadas como em *realçado*. Tudo que representa entrada ou saída do computador, em particular os comandos, código de programa e tela de saída, é mostrado em fonte monoespaçada (`exemplo`). Dentro destas passagens, itálico (*exemplo*) indica um posicionador (*placeholder*); deve ser inserido o valor verdadeiro em seu lugar. Em certas ocasiões, partes do código do programa são enfatizadas em negrito (**exemplo**), se foram adicionadas ou modificadas com relação ao exemplo anterior.

São utilizadas as seguintes convenções na sinopse dos comandos: os colchetes (`[` e `]`) indicam partes opcionais (Na sinopse dos comandos Tcl, são utilizados pontos de interrogação (`?`) em vez dos colchetes, como é usual no Tcl). As chaves (`{` e `}`), e as barras verticais (`|`), indicam que deve ser escolhida uma das alternativas. Os pontos (`...`) significam que o elemento anterior pode ser repetido.

Quando fica mais claro, os comandos SQL são precedidos pelo `prompt =>`, e os comandos para o interpretador de comandos são precedidos pelo `prompt $`. Entretanto, normalmente os `prompts` não são exibidos.

*Administrador* geralmente é a pessoa responsável pela instalação e funcionamento do servidor. *Usuário* pode ser qualquer um usando, ou querendo usar, qualquer parte do sistema PostgreSQL. Estes termos não devem ser interpretados ao pé da letra; este livro não estabelece premissas com relação aos procedimentos do administrador do sistema.

## 4. Outras informações

Além da documentação, ou seja, este livro, existem outras informações sobre o PostgreSQL:

### FAQs

A lista de perguntas freqüentemente formuladas (Frequently Asked Questions - FAQ (<http://www.postgresql.org/docs/faq/>)) contém respostas continuamente atualizadas para as perguntas mais freqüentes.

### READMEs

Os arquivos README (leia-me) estão disponíveis na maioria dos pacotes contribuídos.

### Sítio na Web

O sítio na Web do PostgreSQL (<http://www.postgresql.org>) contém detalhes sobre a última versão e outras informações para tornar o trabalho ou a diversão com o PostgreSQL mais produtiva. Visite também o sítio PostgreSQL-BR, O ponto de informações para brasileiros (<http://www.postgresql.org.br/>).

### Listas de discussão

As listas de discussão (<http://www.postgresql.org/community/lists/>) são bons lugares ter as perguntas respondidas, para trocar experiências com outros usuários, e para fazer contato com os desenvolvedores. Consulte o sítio na Web do PostgreSQL para obter detalhes, ou a lista de discussão postgresql-br - PostgreSQL Brasil (<http://br.groups.yahoo.com/group/postgresql-br/>).

Você mesmo!

O PostgreSQL é um projeto de código aberto. Como tal, depende do apoio permanente da comunidade de usuários. Quando começamos a utilizar o PostgreSQL dependemos da ajuda de outros, tanto através da documentação quanto das listas de discussão. Considere então retribuir seus conhecimentos. Leia as listas de discussão e responda as perguntas. Se você aprender algo que não esteja na documentação, escreva e contribua. Se você adicionar funcionalidades ao código, contribua com estas funcionalidades.

Projeto de Tradução da Documentação do PostgreSQL para o Português do Brasil

Este projeto tem por finalidade traduzir a documentação do PostgreSQL para o Português do Brasil, permitindo a todos os interessados consultar, baixar, colocar em seus próprios sites, distribuir através de CDs, ou de qualquer outra mídia, esta documentação traduzida. Também estão disponíveis os arquivos SGML através do CVS do SourceForge. (<http://cvs.sourceforge.net/viewcvs.py/pgdocptbr/>) Participe! (<http://sourceforge.net/projects/pgdocptbr/>)

## 5. Guia para informar erros

Quando for encontrado algum erro no PostgreSQL desejamos ser informados. Seus relatórios de erro são importantes para tornar o PostgreSQL mais confiável, porque mesmo o cuidado mais extremo não pode garantir que todas as partes do PostgreSQL funcionam em todas as plataformas sob qualquer circunstância.

As sugestões abaixo têm por objetivo ajudá-lo a preparar relatórios de erro que possam ser tratados de forma eficaz. Ninguém é obrigado a segui-las, mas são feitas para serem vantajosas para todos.

Não podemos prometer corrigir todos os erros imediatamente. Se o erro for óbvio, crítico, ou afetar muitos usuários, existe uma boa chance de alguém investigá-lo. Pode acontecer, também, nós solicitarmos que você atualize para uma nova versão, para ver se o erro também acontece na nova versão. Também podemos decidir que o erro não poderá ser corrigido antes de ser feita uma importante reescrita planejada ou, talvez, simplesmente esta correção seja muito difícil e existem assuntos mais importantes na agenda. Se for necessária ajuda imediata, deve ser levada em consideração a contratação de um suporte comercial.

### 5.1. Identificação de erros

Antes de informar um erro, por favor leia e releia a documentação para verificar se realmente pode ser feito o que está se tentando fazer. Se não estiver claro na documentação se pode ou não ser feito, por favor informe isto também; é uma falha na documentação. Se for visto que o programa faz algo diferente do que está especificado na documentação, isto também é um erro. Pode incluir, sem estar restrito, as seguintes circunstâncias:

- O programa termina com um erro fatal, ou com uma mensagem de erro do sistema operacional que aponta para um problema no programa (um exemplo oposto seria uma mensagem de “disco cheio”, porque o próprio usuário deve corrigir este problema).
- O programa produz uma saída errada para uma determinada entrada.
- O programa não aceita uma entrada válida (conforme definido na documentação).
- O programa aceita uma entrada inválida sem enviar uma mensagem de erro. Porém, tenha em mente que a sua idéia de entrada inválida pode ser a nossa idéia de uma extensão, ou de compatibilidade com a prática tradicional.
- Em uma plataforma suportada, a compilação, montagem ou instalação do PostgreSQL, de acordo com as instruções, falha.

Aqui “programa” se refere a qualquer executável, e não apenas ao processo servidor.

Estar lento ou consumir muitos recursos não é necessariamente um erro. Leia a documentação ou faça perguntas em uma lista de discussão pedindo ajuda para ajustar seus aplicativos. Não agir em conformidade com o padrão SQL também não é necessariamente um erro, a não ser que a conformidade com a funcionalidade específica esteja explicitamente informada.

Antes de prosseguir, verifique a lista TODO (a fazer) e a FAQ para ver se o erro já não é conhecido. Se você não conseguir decodificar a informação da lista TODO, relate seu problema. O mínimo que podemos fazer é tornar a lista TODO mais clara.

## 5.2. O que informar

O mais importante a ser lembrado sobre informar erros é declarar todos os fatos, e somente os fatos. Não especule sobre o que você pensa que deu errado, o que “parece que deve ser feito”, ou em que parte do programa está o erro. Se não estiver familiarizado com a implementação você provavelmente vai supor errado, e não vai nos ajudar nem um pouco. E, mesmo que você esteja familiarizado, uma explicação educada é um grande suplemento, mas não substitui os fatos. Se formos corrigir o erro, temos que vê-lo acontecer primeiro. Informar meramente os fatos é relativamente direto (provavelmente pode ser copiado e colado a partir da tela), mas geralmente são deixados de fora detalhes importantes porque se pensou que não tinham importância, ou que o relatório seria entendido de qualquer maneira.

Todo relatório de erro deve conter os seguintes itens:

- A seqüência exata dos passos, *desde o início do programa*, necessários para reproduzir o problema. Isto deve estar autocontido; não é suficiente enviar meramente o comando `SELECT`, sem enviar os comandos `CREATE TABLE` e `INSERT` que o precederam, caso a saída dependa dos dados contidos nas tabelas. Não temos tempo para realizar a engenharia reversa do esquema do seu banco de dados e, se tivermos que criar nossos próprios dados, provavelmente não vamos conseguir reproduzir o problema.

O melhor formato para um caso de teste, para problemas relacionados com a linguagem SQL, é um arquivo mostrando o problema que possa ser executado a partir do utilitário `psql` (certifique-se não existir nada em seu arquivo de inicialização `~/.psqlrc`). Um modo fácil de começar este arquivo é usar o `pg_dump` para gerar as declarações da tabela e dos dados necessários para montar o cenário, e depois adicionar o comando problemático. Incentivamos você a minimizar o tamanho do exemplo, mas isto não é absolutamente necessário. Se o erro for reproduzível, nós o encontraremos de qualquer maneira.

Se o seu aplicativo utiliza alguma outra interface cliente, tal como o PHP, então, por favor, tente isolar o comando problemático. Provavelmente não iremos configurar um servidor Web para reproduzir o seu problema. De qualquer maneira, lembre-se de fornecer os arquivos de entrada exatos, e não suponha que o problema aconteça com “arquivos grandes” ou “bancos de dados de tamanho médio”, etc., porque estas informações são muito pouco precisas para serem úteis.

- A saída recebida. Por favor, não diga que “não funcionou” ou que “deu pau”. Se houver uma mensagem de erro mostre-a, mesmo que você não a entenda. Se o programa terminar com um erro do sistema operacional, diga qual. Se nada acontecer, informe. Mesmo que o resultado do seu caso de teste seja o término anormal do programa, ou seja óbvio de alguma outra forma, pode ser que não aconteça na nossa plataforma. O mais fácil a ser feito é copiar a saída do terminal, se for possível.

**Nota:** Se estiver sendo informada uma mensagem de erro, por favor obtenha a forma mais verbosa da mensagem. No `psql` execute antes `\set VERBOSITY verbose`. Se a mensagem estiver sendo extraída do `log` do servidor, defina o parâmetro em tempo de execução `log_error_verbosity` como `verbose`, para serem registrados todos os detalhes.

**Nota:** No caso de erros fatais, a mensagem de erro informada pelo cliente pode não conter toda a informação disponível. Por favor, olhe também a saída do `log` do servidor de banco de dados. Se você não mantém a saída do `log` do servidor, esta é uma boa hora para começar.

- A saída esperada é uma informação importante a ser declarada. Se for escrito apenas “Este comando produz esta saída” ou “Isto não é o esperado”, poderemos executar, olhar a saída, e achar que está tudo correto, exatamente o que esperávamos que fosse. Não devemos perder tempo decodificando a semântica exata por trás de seus comandos. Abstenha-se, especialmente, de dizer meramente “Isto não é o que o SQL diz ou o que o Oracle faz”. Pesquisar o comportamento correto do SQL não é uma tarefa divertida, nem sabemos como todos os outros bancos de dados relacionais existentes se comportam (se o problema for o término anormal do programa, este item obviamente pode ser omitido).
- Todas as opções de linha de comando e outras opções de inicialização, incluindo as variáveis de ambiente relevantes e os arquivos de configuração que foram alterados em relação ao padrão. Novamente, forneça a informação exata. Se estiver sendo utilizada uma distribuição pré-configurada, que inicializa o servidor de banco de dados durante o `boot`, deve-se tentar descobrir como isto é feito.
- Qualquer coisa feita que seja diferente das instruções de instalação.
- A versão do PostgreSQL. Pode ser executado o comando `SELECT version();` para descobrir a versão do servidor ao qual se está conectado. A maioria dos programas executáveis suportam a opção `--version`; pelo menos `postmaster`

`--version` e `psql --version` devem funcionar. Se a função ou as opções não existirem, então a versão sendo usada é antiga o suficiente para merecer uma atualização. Se estiver sendo usada uma versão pré-configurada, como RPMs, informe, incluindo qualquer sub-versão que o pacote possa ter. Se estiver se referindo a um instantâneo do CVS isto deve ser mencionado, incluindo a data e a hora.

Se a sua versão for anterior a 8.0.0, quase certamente lhe pediremos para atualizar. Existem muitas correções de erro e melhorias a cada nova liberação e, portanto, é bem possível que o erro encontrado em uma versão antiga do PostgreSQL já esteja corrigido. Só podemos oferecer suporte limitado às instalações usando versões antigas do PostgreSQL; se for desejado mais do que pode ser fornecido, deve ser levado em consideração a contratação de um suporte comercial.

- Informações da plataforma. Isto inclui o nome e a versão do núcleo, a biblioteca C, o processador e a memória. Na maioria dos casos é suficiente informar o fornecedor e a versão, mas não se deve supor que todo mundo sabe exatamente o que o “Debian” contém, ou que todo mundo use Pentium. Havendo problemas de instalação, então também são necessárias informações sobre as ferramentas empregadas (compilador, `make`, etc.).

Não tenha medo que seu relatório de erro fique muito longo. Este é um fato da vida. É melhor informar tudo da primeira vez do que termos que extrair os fatos de você. Por outro lado, se seus arquivos de entrada são enormes, é justo perguntar primeiro se alguém está interessado em vê-los.

Não perca seu tempo tentando descobrir que mudanças na entrada fazem o problema desaparecer. Isto provavelmente não vai ajudar a solucionar o problema. Se for visto que o erro não pode ser corrigido imediatamente, você vai ter tempo para descobrir e compartilhar sua descoberta. Também, novamente, não perca seu tempo adivinhando porque o erro existe, nós o descobriremos em breve.

Ao escrever o relatório de erro, por favor escolha uma terminologia que não confunda. O pacote de software em seu todo é chamado de “PostgreSQL” e, algumas vezes, de “Postgres” para encurtar. Se estiver se referindo especificamente ao processo servidor mencione isto, não diga apenas “o PostgreSQL caiu”. A queda de um único processo servidor é bem diferente da queda do processo “postmaster” pai; por favor não diga “o postmaster caiu” quando um único processo servidor caiu, nem o contrário. Além disso os programas cliente, como o cliente interativo “psql”, são completamente separados do servidor. Por favor, tente especificar se o problema está no lado cliente ou no lado servidor.

### 5.3. Onde informar os erros

De modo geral, envie relatórios de erro para a lista de discussão de relatórios de erros em `<pgsql-bugs@postgresql.org>`. É necessário utilizar um assunto descritivo para a mensagem de correio eletrônico, talvez partes da própria mensagem de erro.

Outra forma é preencher o relatório de erro disponível no sítio do projeto na Web em <http://www.postgresql.org/>. Preencher o relatório de erro desta forma faz com que seja enviado para a lista de discussão `<pgsql-bugs@postgresql.org>`.

Não envie o relatório de erro para nenhuma lista de discussão dos usuários, tal como `<pgsql-sql@postgresql.org>` ou `<pgsql-general@postgresql.org>`. Estas listas de discussão são para responder as perguntas dos usuários, e seus assinantes normalmente não desejam receber relatórios de erro. Mais importante ainda, eles provavelmente não vão conseguir corrigir o erro.

Por favor, também *não* envie relatórios para a lista de discussão dos desenvolvedores em `<pgsql-hackers@postgresql.org>`. Esta lista é para discutir o desenvolvimento do PostgreSQL, e gostamos de manter os relatórios de erro em separado. Podemos decidir discutir seu relatório de erro em `pgsql-hackers`, se o problema necessitar uma melhor averiguação.

Se você tiver problema com a documentação, o melhor lugar para informar é na lista de discussão da documentação em `<pgsql-docs@postgresql.org>`.<sup>4</sup> Por favor seja específico sobre qual parte da documentação você não está satisfeito.

Se seu erro for um problema de portabilidade em uma plataforma não suportada, envie uma mensagem de correio eletrônico para `<pgsql-ports@postgresql.org>`, para que nós (e você) possamos trabalhar para portar o PostgreSQL para esta plataforma.

**Nota:** Por causa da grande quantidade de *spam* na Internet, todos os endereços de correio eletrônico acima são de listas de discussão fechadas, ou seja, primeiro é necessário assinar a lista para depois poder enviar mensagens (entretanto, não é necessário assinar nenhuma lista para utilizar o formulário de relatório de erro da Web). Se você deseja enviar uma mensagem de correio eletrônico, mas não deseja receber o tráfego da lista, você pode subscrever e

configurar sua opção de subscrição com `nomail`. Para maiores informações envie uma mensagem para `<majordomo@postgresql.org>` contendo apenas a palavra `help` no corpo da mensagem.

## Notas

1. Um SGBDOR incorpora as tecnologias de orientação a objeto e relacional. A maioria dos produtos adere ao padrão SQL:1999, mas alguns implementam um enfoque proprietário. Em sua essência, o modelo de objetos do SQL:1999 possui as mesmas funcionalidades do modelo de objetos usado pelos sistemas gerenciadores de banco de dados orientados a objeto (SGBDOO), mas de uma forma diferente da maioria dos SGBDOO. Isto se deve à obrigação do SQL:1999 ser compatível com o SQL-92. Esta obrigação fez com que o modelo de objetos do SQL:1999 fosse adaptado ao modelo relacional do SQL-92. Como resultado, o modelo de objetos do SQL:1999 não corresponde ao modelo de objetos usado pelas linguagens de programação orientadas a objeto. Desta forma, o termo “orientado a objeto” não pode ser usado para descrever este modelo, porque implicaria que o modelo do banco de dados correspondesse ao modelo da programação orientada a objeto. Em seu lugar, usa-se o termo “objeto-relacional”. ORDBMS articles and products (<http://www.service-architecture.com/ordbms/>) (N. do T.)
2. Outros produtos que se enquadram nesta categoria são o Oracle 8, o DB2 da IBM e o Illustra da Informix. (N. do T.)
3. A implementação da inversão de objetos grandes divide os objetos grandes em “pedaços”, e armazena os pedaços em linhas no banco de dados. Um índice `B-tree` garante a procura rápida do número correto do pedaço ao serem feitos acessos aleatórios de leitura e gravação. Manual de Referência do PostgreSQL 6.3 (N. do T.)
4. Por favor, informe os erros de tradução encontrados nesta documentação para `<halley@users.sourceforge.net>`. (N. do T.)

# I. Tutorial

Bem vindo ao Tutorial do PostgreSQL. Os poucos capítulos a seguir têm por objetivo fornecer uma introdução simples ao PostgreSQL, aos conceitos de banco de dados relacional e à linguagem SQL, para os iniciantes em qualquer um destes tópicos. Somente é pressuposto um conhecimento geral sobre a utilização de computadores. Nenhuma experiência com Unix ou em programação é necessária. Esta parte tem como objetivo principal fornecer experiência prática sobre os aspectos importantes do PostgreSQL. Não há nenhuma intenção em dar-se um tratamento completo ou abrangente dos tópicos cobertos.

Após ler este tutorial pode-se prosseguir através da leitura da Parte II para obter um maior conhecimento formal da linguagem SQL, ou da Parte IV para obter informações sobre o desenvolvimento de aplicativos para o PostgreSQL. Aqueles que instalam e gerenciam seus próprios servidores também devem ler a Parte III.

# Capítulo 1. Início

## 1.1. Instalação

Para que se possa usar o PostgreSQL é necessário instalá-lo, obviamente. É possível que o PostgreSQL já esteja instalado na máquina, seja porque está incluído na distribuição do sistema operacional <sup>1</sup>, ou porque o administrador do sistema fez a instalação. Se este for o caso, devem ser obtidas informações na documentação do sistema operacional, ou com o administrador do sistema, sobre como acessar o PostgreSQL.

Não havendo certeza se o PostgreSQL está disponível, ou se pode ser utilizado para seus experimentos, então você mesmo poderá fazer a instalação. Proceder desta maneira não é difícil, podendo ser um bom exercício. O PostgreSQL pode ser instalado por qualquer usuário sem privilégios, porque não é necessário nenhum acesso de superusuário (root).

Se for instalar o PostgreSQL por si próprio, então leia o Capítulo 14 para conhecer as instruções de instalação, e depois retorne para este guia quando a instalação estiver terminada. Certifique-se de seguir de perto a seção sobre a configuração das variáveis de ambiente apropriadas.

Se o administrador do sistema não fez a configuração da maneira padrão, talvez seja necessário algum trabalho adicional. Por exemplo, se a máquina servidora de banco de dados for uma máquina remota, será necessário definir a variável de ambiente `PGHOST` com o nome da máquina servidora de banco de dados. Também, talvez tenha que ser definida a variável de ambiente `PGPORT`. A regra básica é esta: quando se tenta iniciar um programa aplicativo e este informa que não está conseguindo conectar com o banco de dados, deve ser consultado o administrador do servidor ou, caso seja você mesmo, a documentação, para ter certeza que o ambiente está configurado de maneira apropriada. Caso não tenha entendido o parágrafo anterior então, por favor, leia a próxima seção.

## 1.2. Fundamentos da arquitetura

Antes de prosseguir, é necessário conhecer a arquitetura de sistema básica do PostgreSQL. Compreender como as partes do PostgreSQL interagem torna este capítulo mais claro.

No jargão de banco de dados, o PostgreSQL utiliza o modelo cliente-servidor. Uma sessão do PostgreSQL consiste nos seguintes processos (programas) cooperando entre si:

- Um processo servidor, que gerencia os arquivos de banco de dados, recebe conexões dos aplicativos cliente com o banco de dados, e executa ações no banco de dados em nome dos clientes. O programa servidor de banco de dados se chama `postmaster`.
- O aplicativo cliente do usuário (`frontend`) que deseja executar operações de banco de dados. Os aplicativos cliente podem ter naturezas muito diversas: o cliente pode ser uma ferramenta no modo caractere, um aplicativo gráfico, um servidor Web que acessa o banco de dados para mostrar páginas Web, ou uma ferramenta especializada para manutenção do banco de dados. Alguns aplicativos cliente são fornecidos na distribuição do PostgreSQL, sendo a maioria desenvolvido pelos usuários. <sup>2</sup>

Como é típico em aplicações cliente-servidor, o cliente e o servidor podem estar em hospedeiros diferentes. Neste caso se comunicam através de uma conexão de rede TCP/IP. Deve-se ter isto em mente, porque arquivos que podem ser acessados na máquina cliente podem não ser acessíveis pela máquina servidora (ou somente podem ser acessados usando um nome de arquivo diferente).

O servidor PostgreSQL pode tratar várias conexões simultâneas de clientes. Para esta finalidade é iniciado um novo processo (`fork` <sup>3</sup>) para cada conexão. Deste ponto em diante, o cliente e o novo processo servidor se comunicam sem intervenção do processo `postmaster` original. Portanto, o `postmaster` está sempre executando aguardando por novas conexões dos clientes, enquanto os clientes e seus processos servidor associados surgem e desaparecem (obviamente tudo isso é invisível para o usuário, sendo mencionado somente para ficar completo).

## 1.3. Criação de banco de dados

O primeiro teste para verificar se é possível acessar o servidor de banco de dados é tentar criar um banco de dados. Um servidor PostgreSQL pode gerenciar muitos bancos de dados. Normalmente é utilizado um banco de dados em separado para cada projeto ou para cada usuário.

Possivelmente, o administrador já criou um banco de dados para seu uso. Ele deve ter dito qual é o nome do seu banco de dados. Neste caso esta etapa pode ser omitida, indo-se direto para a próxima seção.

Para criar um novo banco de dados, chamado `meu_bd` neste exemplo, deve ser utilizado o comando:

```
$ createdb meu_bd
```

Que deve produzir a seguinte resposta:

```
CREATE DATABASE
```

Se esta resposta for mostrada então esta etapa foi bem sucedida, podendo-se pular o restante da seção .

Se for mostrada uma mensagem semelhante a

```
createdb: comando não encontrado
```

então o PostgreSQL não foi instalado da maneira correta, ou não foi instalado, ou o caminho de procura não foi definido corretamente. Tente executar o comando utilizando o caminho absoluto:

```
$ /usr/local/pgsql/bin/createdb meu_bd
```

O caminho na sua máquina pode ser diferente. <sup>4</sup> Fale com o administrador, ou verifique novamente as instruções de instalação para corrigir a situação.

Outra resposta pode ser esta: <sup>5</sup>

```
createdb: could not connect to database template1: could not connect to server:
No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

#### -- Tradução da mensagem (N. do T.)

```
createdb: não foi possível conectar ao banco de dados template1
        : não foi possível conectar ao servidor:
Arquivo ou diretório inexistente
        O servidor está executando localmente e aceitando
        conexões no soquete do domínio Unix "/tmp/.s.PGSQL.5432"?
```

Significando que o servidor não foi inicializado, ou que não foi inicializado onde o `createdb` esperava que fosse. Novamente, verifique as instruções de instalação ou consulte o administrador.

Outra resposta pode ser esta:

```
createdb: could not connect to database template1:
FATAL: user "joel" does not exist
```

#### -- Tradução da mensagem (N. do T.)

```
createdb: não foi possível conectar ao banco de dados template1:
FATAL: o usuário "joel" não existe
```

onde é mencionado o seu próprio nome de login. Isto vai acontecer se o administrador não tiver criado uma conta de usuário no PostgreSQL para seu uso (As contas de usuário do PostgreSQL são distintas das contas de usuário do sistema operacional). Se você for o administrador, obtenha ajuda para criar contas no Capítulo 17. Será necessário se tornar o usuário do sistema operacional que instalou o PostgreSQL (geralmente `postgres`) para criar a primeira conta de usuário. Também pode ter sido atribuído para você um nome de usuário do PostgreSQL diferente do nome de usuário do sistema operacional; neste caso, é necessário utilizar a chave `-U`, ou definir a variável de ambiente `PGUSER`, para especificar o nome de usuário do PostgreSQL.

Caso se possua uma conta de usuário, mas esta conta não possua o privilégio necessário para criar bancos de dados, será exibida a seguinte mensagem:



```
createdb: database creation failed:
ERROR: permission denied to create database
```

#### -- Tradução da mensagem (N. do T.)

```
createdb: a criação do banco de dados falhou:
ERRO: negada a permissão para criar banco de dados
```

Nem todo usuário possui autorização para criar bancos de dados. Se o PostgreSQL não permitir criar o banco de dados, então o administrador deve conceder permissão para você criar bancos de dados. Consulte o administrador caso isto ocorra. Caso tenha instalado o PostgreSQL por si próprio, então conecte usando a mesma conta de usuário utilizada para inicializar o servidor, para as finalidades deste tutorial.<sup>6</sup>

Também podem ser criados bancos de dados com outros nomes. O PostgreSQL permite a criação de qualquer número de bancos de dados em uma instalação. Os nomes dos bancos de dados devem ter o primeiro caractere alfabético, sendo limitados a um comprimento de 63 caracteres. Uma escolha conveniente é criar o banco de dados com o mesmo nome do usuário corrente. Muitas ferramentas assumem este nome de banco de dados como sendo o nome padrão, evitando a necessidade de digitá-lo. Para criar este banco de dados deve ser digitado simplesmente:

```
$ createdb
```

Caso não deseje mais utilizar o seu banco de dados, pode removê-lo. Por exemplo, se você for o dono (criador) do banco de dados `meu_bd`, poderá removê-lo utilizando o seguinte comando:

```
$ dropdb meu_bd
```

Para este comando o nome da conta não é utilizado como nome padrão do banco de dados: o nome sempre deve ser especificado. Esta ação remove fisicamente todos os arquivos associados ao banco de dados não podendo ser desfeita, portanto esta operação somente deve ser feita após um longo período de reflexão.

Podem ser encontradas informações adicionais sobre os comandos `createdb` e `dropdb` em `createdb` e `dropdb`, respectivamente.

## 1.4. Acesso a banco de dados

Após o banco de dados ter sido criado, este pode ser acessado pela:

- Execução do programa de terminal interativo do PostgreSQL chamado *psql*, que permite entrar, editar e executar comandos SQL interativamente.
- Utilização de uma ferramenta cliente gráfica existente como o PgAccess, ou de um pacote de automação de escritórios com suporte a ODBC para criar e manusear bancos de dados. Estas possibilidades não estão descritas neste tutorial.
- Criação de aplicativos personalizados, usando um dos vários vínculos com linguagens disponíveis. Estas possibilidades são mostradas mais detalhadamente na Parte IV.

Você provavelmente vai desejar ativar o `psql` para executar os exemplos deste tutorial. O `psql` pode ser ativado para usar o banco de dados `meu_bd` digitando o comando:

```
$ psql meu_bd
```

Se o nome do banco de dados for omitido, então será usado o nome padrão igual ao nome da conta do usuário. Isto já foi visto na seção anterior.

O `psql` saúda o usuário com a seguinte mensagem:

```
Welcome to psql 8.0.0, the PostgreSQL interactive terminal.
```

```
Type: \copyright for distribution terms
      \h for help with SQL commands
      \? for help on internal slash commands
      \g or terminate with semicolon to execute query
      \q to quit
```

```
meu_bd=>
```

**-- Tradução da mensagem (N. do T.)**

Bem-vindo ao `psql` 8.0.0, o terminal interativo do PostgreSQL.

```
Digite: \copyright para mostrar a licença da distribuição
        \h para ajuda nos comandos SQL
        \? para ajuda nos comandos de contrabarra internos
        \g ou finalizar com ponto-e-vírgula para executar o comando
        \q para sair
```

```
meu_bd=>
```

A última linha também pode ser

```
meu_bd=#
```

significando que o usuário é um superusuário do banco de dados, acontecendo geralmente quando se instala o PostgreSQL por si próprio. Ser um superusuário significa não estar sujeito a controles de acesso. Para as finalidades deste tutorial isto não tem importância.

Caso aconteçam problemas ao inicializar o `psql`, então retorne à seção anterior. Os diagnósticos do `psql` e do `createdb` são semelhantes, e se um funcionou o outro deve funcionar também.

A última linha exibida pelo `psql` é o `prompt`, indicando que o `psql` está lhe aguardando, e que você pode digitar comandos SQL dentro do espaço de trabalho mantido pelo `psql`. Tente estes comandos:

```
meu_bd=> SELECT version();
```

```

                version
-----
PostgreSQL 8.0.0 on i586-pc-linux-gnu, compiled by GCC 2.96
(1 linha)
```

```
meu_bd=> SELECT current_date;
```

```

        date
-----
2005-05-17
(1 linha)
```

```
meu_bd=> SELECT 2 + 2;
```

```

?column?
-----
4
(1 linha)
```

O programa `psql` possui vários comandos internos que não são comandos SQL. Eles começam pelo caractere de contrabarra, “\”. Alguns destes comandos são mostrados na mensagem de boas vindas. Por exemplo, pode ser obtida ajuda sobre a sintaxe de vários comandos SQL do PostgreSQL digitando:

```
meu_bd=> \h
```

Para sair do `psql` digite

```
meu_bd=> \q
```

e o `psql` terminará retornando para o interpretador de comandos (para conhecer outros comandos internos digite `\?` no `prompt` do `psql`). Todas as funcionalidades do `psql` estão documentadas em `psql`. Se o PostgreSQL tiver sido instalado corretamente, também pode-se digitar `man psql` na linha de comando do sistema operacional para ver a documentação. Neste tutorial não utilizaremos estas funcionalidades explicitamente, mas use por si próprio quando julgar adequado.

## Notas

1. *distribuição* — Uma árvore de código fonte de `software` empacotada para distribuição; Desde cerca de 1996 a utilização não qualificada deste termo geralmente implica numa “distribuição Linux”. A forma curta “distro” geralmente é utilizada neste sentido. The Jargon File (<http://www.catb.org/~esr/jargon/html/D/distribution.html>) (N. do T.)
2. Também pode ser utilizada a ferramenta de administração do PostgreSQL baseada na Web `phpPgAdmin` (<http://phppgadmin.sourceforge.net/>). (N. do T.)
3. *fork* — Para criar um novo processo, o processo copia a si próprio através da chamada de sistema `fork`. O `fork` cria uma cópia do processo original que é em grande parte idêntica à ancestral. O novo processo possui um PID (identificador de processo) próprio, e suas próprias informações de contabilização. O `fork` possui a propriedade única de retornar dois valores diferentes. Do ponto de vista do filho retorna zero. Por outro lado, para o pai é retornado o PID do filho recém criado. Uma vez que fora isso os dois processos são idênticos, ambos precisam examinar o valor retornado para descobrir o papel a ser desempenhado. *Linux Administration Handbook*, Evi Nemeth e outros, Prentice Hall, 2002. (N. do T.)
4. `/usr/bin/createdb` no RedHat, Fedora, Mandrake e Debian. (N. do T.)
5. *soquete* — As chamadas de sistema para estabelecer a conexão são um tanto diferentes para o cliente e para o servidor, mas ambas envolvem basicamente a construção de um soquete. O soquete é uma extremidade do canal de comunicação entre processos. Cada um dos dois processos estabelece seu próprio soquete. Para criar o soquete, o programa precisa especificar o domínio de endereço e o tipo de soquete. Dois processos podem se comunicar somente se seus soquetes são do mesmo tipo e do mesmo domínio. Existem dois domínios de endereço amplamente utilizados: o domínio Unix, no qual dois processos compartilham um arquivo do sistema em comum para se comunicar; e o domínio Internet, no qual dois processos executando em dois hospedeiros quaisquer na Internet se comunicam. Cada um destes domínios possui seu próprio formato de endereço. O endereço de soquete do domínio Unix é uma cadeia de caracteres, que é basicamente uma entrada no sistema de arquivos. O endereço de soquete do domínio Internet consiste no endereço de Internet da máquina hospedeira (todo computador na Internet possui um endereço único de 32 bits, geralmente chamado de endereço de IP). Adicionalmente, no domínio Internet, cada soquete necessita do número da porta no hospedeiro. *Sockets Tutorial* (<http://people.cs.uchicago.edu/%7Emark/51081/labs/LAB6/sock.html>) (N. do T.)
6. Uma explicação do motivo pelo qual isto funciona: Os nomes de usuário do PostgreSQL são distintos das contas de usuário do sistema operacional. Ao estabelecer a conexão com um banco de dados, pode ser escolhido o nome do usuário do PostgreSQL com o qual se deseja fazer a conexão; Se isto não for feito, o padrão é utilizar um nome igual ao da conta atual do sistema operacional. Como isto ocorre, sempre existirá uma conta de usuário do PostgreSQL que possui o nome igual ao do usuário do sistema operacional que inicializou o servidor; acontece, também, que este usuário sempre tem permissão para criar banco de dados. Em vez de conectar como este usuário, pode ser especificada a opção `-U` em todos os aplicativos para escolher o nome do usuário do PostgreSQL com o qual se deseja conectar.

# Capítulo 2. A linguagem SQL

## 2.1. Introdução

Este capítulo fornece uma visão geral sobre como utilizar a linguagem SQL para realizar operações simples. O propósito deste tutorial é apenas fazer uma introdução e, de forma alguma, ser um tutorial completo sobre a linguagem SQL. Existem muitos livros escritos sobre a linguagem SQL, incluindo *Understanding the New SQL* e *A Guide to the SQL Standard*. É preciso estar ciente que algumas funcionalidades da linguagem SQL do PostgreSQL são extensões ao padrão.

Nos exemplos a seguir supõe-se que tenha sido criado o banco de dados chamado `meu_bd`, conforme descrito no capítulo anterior, e que o `psql` esteja ativo.

Os exemplos presentes neste manual também podem ser encontrados na distribuição do código fonte do PostgreSQL, no diretório `src/tutorial/`.<sup>1</sup> Para usar estes arquivos, primeiro deve-se tornar o diretório `src/tutorial/` o diretório corrente, e depois executar o utilitário `make`, conforme mostrado abaixo:

```
$ cd ../src/tutorial
$ make
```

Este procedimento cria os scripts e compila os arquivos C contendo as funções e tipos definidos pelo usuário (Deve ser utilizado o `make` do GNU neste procedimento, que pode ter um nome diferente no sistema sendo utilizado, geralmente `gmake`).<sup>2</sup> Depois disso, para iniciar o tutorial faça o seguinte:

```
$ cd ../src/tutorial
$ psql -s meu_bd
...
```

```
meu_bd=> \i basics.sql
```

O comando `\i` lê os comandos no arquivo especificado. A opção `-s` ativa o modo passo a passo, que faz uma pausa antes de enviar cada comando para o servidor. Os comandos utilizados nesta seção estão no arquivo `basics.sql` (`./basics.sql`).

## 2.2. Conceitos

O PostgreSQL é um *sistema de gerenciamento de banco de dados relacional* (SGBDR). Isto significa que é um sistema para gerenciar dados armazenados em *relações*. Relação é, essencialmente, um termo matemático para *tabela*. A noção de armazenar dados em tabelas é tão trivial hoje em dia que pode parecer totalmente óbvio, mas existem várias outras formas de organizar bancos de dados. Arquivos e diretórios em sistemas operacionais tipo Unix são um exemplo de banco de dados hierárquico. Um desenvolvimento mais moderno são os bancos de dados orientados a objeto.

Cada tabela é uma coleção nomeada de *linhas*. Todas as linhas de uma determinada tabela possuem o mesmo conjunto de *colunas* nomeadas, e cada coluna é de um tipo de dado específico. Enquanto as colunas possuem uma ordem fixa nas linhas, é importante lembrar que o SQL não garante a ordem das linhas dentro de uma tabela (embora as linhas possam ser explicitamente ordenadas para a exibição).

As tabelas são agrupadas em bancos de dados, e uma coleção de bancos de dados gerenciados por uma única instância do servidor PostgreSQL forma um *agrupamento* de bancos de dados.

## 2.3. Criação de tabelas

Pode-se criar uma tabela especificando o seu nome juntamente com os nomes das colunas e seus tipos de dado:

```
CREATE TABLE clima (
    cidade          varchar(80),
    temp_min        int,          -- temperatura mínima
    temp_max        int,          -- temperatura máxima
    prcp            real,         -- precipitação
    data            date
);
```

Este comando pode ser digitado no `psql` com quebras de linha. O `psql` reconhece que o comando só termina quando é encontrado o ponto-e-vírgula.

Espaços em branco (ou seja, espaços, tabulações e novas linhas) podem ser utilizados livremente nos comandos SQL. Isto significa que o comando pode ser digitado com um alinhamento diferente do mostrado acima, ou mesmo tudo em uma única linha. Dois hífen (“--”) iniciam um comentário; tudo que vem depois é ignorado até o final da linha. A linguagem SQL não diferencia letras maiúsculas e minúsculas nas palavras chave e nos identificadores, a não ser que os identificadores sejam colocados entre aspas (“”) para preservar letras maiúsculas e minúsculas, o que não foi feito acima.

No comando, `varchar(80)` especifica um tipo de dado que pode armazenar cadeias de caracteres arbitrárias com comprimento até 80 caracteres; `int` é o tipo inteiro normal; `real` é o tipo para armazenar números de ponto flutuante de precisão simples; `date` é o tipo para armazenar data e hora (a coluna do tipo `date` pode se chamar `date`, o que tanto pode ser conveniente quanto pode causar confusão).

O PostgreSQL suporta os tipos SQL padrão `int`, `smallint`, `real`, `double precision`, `char(N)`, `varchar(N)`, `date`, `time`, `timestamp` e `interval`, assim como outros tipos de utilidade geral, e um conjunto abrangente de tipos geométricos. O PostgreSQL pode ser personalizado com um número arbitrário de tipos definidos pelo usuário. Como consequência, sintaticamente os nomes dos tipos não são palavras chave, exceto onde for requerido para suportar casos especiais do padrão SQL.

No segundo exemplo são armazenadas cidades e suas localizações geográficas associadas:

```
CREATE TABLE cidades (
    nome          varchar(80),
    localizacao   point
);
```

O tipo `point` é um exemplo de tipo de dado específico do PostgreSQL.

Para terminar deve ser mencionado que, quando a tabela não é mais necessária, ou se deseja recriá-la de uma forma diferente, é possível removê-la por meio do comando:

```
DROP TABLE nome_da_tabela;
```

## 2.4. Inserção de linhas em tabelas

É utilizado o comando `INSERT` para inserir linhas nas tabelas:

```
INSERT INTO clima VALUES ('São Francisco', 46, 50, 0.25, '1994-11-27');
```

Repare que todos os tipos de dado possuem formato de entrada de dados bastante óbvios. As constantes, que não são apenas valores numéricos, geralmente devem estar entre apóstrofes ('), como no exemplo acima. O tipo `date` é, na verdade, muito flexível em relação aos dados que aceita, mas para este tutorial vamos nos fixar no formato sem ambigüidade mostrado acima.

O tipo `point` requer um par de coordenadas como entrada, como mostrado abaixo:

```
INSERT INTO cidades VALUES ('São Francisco', '(-194.0, 53.0)');
```

A sintaxe usada até agora requer que seja lembrada a ordem das colunas. Uma sintaxe alternativa permite declarar as colunas explicitamente:

```
INSERT INTO clima (cidade, temp_min, temp_max, prcp, data)
VALUES ('São Francisco', 43, 57, 0.0, '1994-11-29');
```

Se for desejado, pode-se declarar as colunas em uma ordem diferente, e pode-se, também, omitir algumas colunas. Por exemplo, se a precipitação não for conhecida:

```
INSERT INTO clima (data, cidade, temp_max, temp_min)
VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Muitos desenvolvedores consideram declarar explicitamente as colunas um estilo melhor que confiar na ordem implícita.

Por favor, entre todos os comando mostrados acima para ter alguns dados para trabalhar nas próximas seções.

Também pode ser utilizado o comando `COPY` para carregar uma grande quantidade de dados a partir de arquivos texto puro. Geralmente é mais rápido, porque o comando `COPY` é otimizado para esta finalidade, embora possua menos flexibilidade que o comando `INSERT`. Para servir de exemplo:

```
COPY clima FROM '/home/user/clima.txt';
```

O arquivo contendo os dados deve poder ser acessado pelo servidor e não pelo cliente, porque o servidor lê o arquivo diretamente. Podem ser obtidas mais informações sobre o comando `COPY` em `COPY`.

## 2.5. Consultar tabelas

Para trazer os dados de uma tabela, a tabela deve ser *consultada*. Para esta finalidade é utilizado o comando `SELECT` do SQL. Este comando é dividido em *lista de seleção* (a parte que especifica as colunas a serem trazidas), *lista de tabelas* (a parte que especifica as tabelas de onde os dados vão ser trazidos), e uma *qualificação opcional* (a parte onde são especificadas as restrições). Por exemplo, para trazer todas as linhas da tabela `clima` digite:

```
SELECT * FROM clima;
```

(aqui `*` é uma forma abreviada de “todas as colunas”).<sup>3</sup> Seriam obtidos os mesmos resultados usando:

```
SELECT cidade, temp_min, temp_max, prcp, data FROM clima;
```

A saída deve ser:

cidade	temp_min	temp_max	prcp	data
São Francisco	46	50	0.25	1994-11-27
São Francisco	43	57	0	1994-11-29
Hayward	37	54		1994-11-29

(3 linhas)

Na lista de seleção podem ser especificadas expressões, e não apenas referências a colunas. Por exemplo, pode ser escrito

```
SELECT cidade, (temp_max+temp_min)/2 AS temp_media, data FROM clima;
```

devendo produzir:

cidade	temp_media	data
São Francisco	48	1994-11-27
São Francisco	50	1994-11-29
Hayward	45	1994-11-29

(3 linhas)

Perceba que a cláusula `AS` foi utilizada para mudar o nome da coluna de saída (a cláusula `AS` é opcional).

A consulta pode ser “qualificada”, adicionando a cláusula `WHERE` para especificar as linhas desejadas. A cláusula `WHERE` contém expressões booleanas (valor verdade), e somente são retornadas as linhas para as quais o valor da expressão booleana for verdade. São permitidos os operadores booleanos usuais (`AND`, `OR` e `NOT`) na qualificação. Por exemplo, o comando abaixo retorna os registros do clima de São Francisco nos dias de chuva:

```
SELECT * FROM clima
WHERE cidade = 'São Francisco' AND prcp > 0.0;
```

Resultado:

cidade	temp_min	temp_max	prcp	data
São Francisco	46	50	0.25	1994-11-27

(1 linha)

Pode ser solicitado que os resultados da consulta sejam retornados em uma determinada ordem:

```
SELECT * FROM clima
ORDER BY cidade;
```

cidade	temp_min	temp_max	prcp	data
Hayward	37	54		1994-11-29
São Francisco	43	57	0	1994-11-29
São Francisco	46	50	0.25	1994-11-27

Neste exemplo a ordem de classificação não está totalmente especificada e, portanto, as linhas de São Francisco podem retornar em qualquer ordem. Mas sempre seriam obtidos os resultados mostrados acima se fosse executado:

```
SELECT * FROM clima
ORDER BY cidade, temp_min;
```

Pode ser solicitado que as linhas duplicadas sejam removidas do resultado da consulta: <sup>4</sup>

```
SELECT DISTINCT cidade
FROM clima;
```

```
cidade
-----
Hayward
São Francisco
(2 linhas)
```

Novamente, neste exemplo a ordem das linhas pode variar. Pode-se garantir resultados consistentes utilizando `DISTINCT` e `ORDER BY` juntos: <sup>5 6</sup>

```
SELECT DISTINCT cidade
FROM clima
ORDER BY cidade;
```

## 2.6. Junções entre tabelas

Até agora as consultas somente acessaram uma tabela de cada vez. As consultas podem acessar várias tabelas de uma vez, ou acessar a mesma tabela de uma maneira que várias linhas da tabela sejam processadas ao mesmo tempo. A consulta que acessa várias linhas da mesma tabela, ou de tabelas diferentes, de uma vez, é chamada de consulta de *junção*. Como exemplo, suponha que se queira listar todas as linhas de clima junto com a localização da cidade associada. Para se fazer isto, é necessário comparar a coluna `cidade` de cada linha da tabela `clima` com a coluna `nome` de todas as linhas da tabela `cidadaes`, e selecionar os pares de linha onde estes valores são correspondentes.

**Nota:** Este é apenas um modelo conceitual, a junção geralmente é realizada de uma maneira mais eficiente que comparar de verdade cada par de linhas possível, mas isto não é visível para o usuário.

Esta operação pode ser efetuada por meio da seguinte consulta:

```
SELECT *
FROM clima, cidades
WHERE cidade = nome;
```

cidade	temp_min	temp_max	prcp	data	nome	localizacao
São Francisco	46	50	0.25	1994-11-27	São Francisco	(-194,53)
São Francisco	43	57	0	1994-11-29	São Francisco	(-194,53)

(2 linhas)

Duas coisas devem ser observadas no resultado produzido:

- Não existe nenhuma linha para a cidade Hayward. Isto acontece porque não existe entrada correspondente na tabela `climas` para Hayward, e a junção ignora as linhas da tabela `climas` sem correspondência. Veremos em breve como isto pode ser mudado.
- Existem duas colunas contendo o nome da cidade, o que está correto porque a lista de colunas das tabelas `climas` e `climas` estão concatenadas. Na prática isto não é desejado, sendo preferível, portanto, escrever a lista das colunas de saída explicitamente em vez de utilizar o `*`:

```
SELECT cidade, temp_min, temp_max, prcp, data, localizacao
FROM climas, cidades
WHERE cidade = nome;
```

**Exercício:** Descobrir a semântica desta consulta quando a cláusula `WHERE` é omitida.

Como todas as colunas possuem nomes diferentes, o analisador encontra automaticamente a tabela que a coluna pertence, mas é um bom estilo qualificar completamente os nomes das colunas nas consultas de junção:

```
SELECT climas.cidade, climas.temp_min, climas.temp_max,
       climas.prcp, climas.data, cidades.localizacao
FROM climas, cidades
WHERE cidades.nome = climas.cidade;
```

As consultas de junção do tipo visto até agora também poderiam ser escritas da seguinte forma alternativa:

```
SELECT *
FROM climas INNER JOIN cidades ON (climas.cidade = cidades.nome);
```

A utilização desta sintaxe não é tão comum quanto a usada acima, mas é mostrada para ajudar a entender os próximos tópicos.

Agora vamos descobrir como se faz para obter as linhas de Hayward. Desejamos o seguinte: que a consulta varra a tabela `climas` e, para cada uma de suas linhas, encontre a linha correspondente na tabela `climas`. Se não for encontrada nenhuma linha correspondente, desejamos que sejam colocados “valores vazios” nas colunas da tabela `climas`. Este tipo de consulta é chamada de *junção externa* (`outer join`). As consultas vistas até agora são junções internas (`inner join`). O comando então fica assim:

```
SELECT *
FROM climas LEFT OUTER JOIN cidades ON (climas.cidade = cidades.nome);
```

cidade	temp_min	temp_max	prcp	data	nome	localizacao
Hayward	37	54		1994-11-29		
São Francisco	46	50	0.25	1994-11-27	São Francisco	(-194,53)
São Francisco	43	57	0	1994-11-29	São Francisco	(-194,53)

(3 linhas)

Esta consulta é chamada de *junção externa esquerda* (`left outer join`), porque a tabela mencionada à esquerda do operador de junção terá cada uma de suas linhas aparecendo na saída pelo menos uma vez, enquanto a tabela à direita terá somente as linhas correspondendo a alguma linha da tabela à esquerda aparecendo na saída. Ao listar uma linha da tabela à esquerda, para a qual não existe nenhuma linha correspondente na tabela à direita, são colocados valores vazios (`null`) nas colunas da tabela à direita.

**Exercício:** Existem também a junção externa direita (`right outer join`) e a junção externa completa (`full outer join`). Tente descobrir o que fazem.

Também é possível fazer a junção da tabela consigo mesma. Isto é chamado de *autojunção* (`self join`). Como exemplo, suponha que desejamos descobrir todas as linhas de clima que estão no intervalo de temperatura de outros registros de clima. Para isso é necessário comparar as colunas `temp_min` e `temp_max` de cada registro de `climas` com as colunas `temp_min` e `temp_max` de todos os outros registros da tabela `climas`, o que pode ser feito utilizando a seguinte consulta:

```
SELECT C1.cidade, C1.temp_min AS menor, C1.temp_max AS maior,
       C2.cidade, C2.temp_min AS menor, C2.temp_max AS maior
FROM climas C1, climas C2
```



```
WHERE C1.temp_min < C2.temp_min
AND C1.temp_max > C2.temp_max;
```

cidade	menor	maior	cidade	menor	maior
São Francisco	43	57	São Francisco	46	50
Hayward	37	54	São Francisco	46	50

(2 linhas)

A tabela clima teve seu nome mudado para C1 e C2, para permitir distinguir o lado esquerdo do lado direito da junção. Estes tipos de “aliases” também podem ser utilizados em outras consultas para reduzir a digitação como, por exemplo:

```
SELECT *
FROM clima w, cidades c
WHERE w.cidade = c.nome;
```

Será vista esta forma de abreviar com bastante frequência.

## 2.7. Funções de agregação

Como a maioria dos produtos de banco de dados relacional, o PostgreSQL suporta funções de agregação. Uma função de agregação computa um único resultado para várias linhas de entrada. Por exemplo, existem funções de agregação para contar (count), somar (sum), calcular a média (avg), o valor máximo (max) e o valor mínimo (min) para um conjunto de linhas.

Para servir de exemplo, é possível encontrar a maior temperatura mínima ocorrida em qualquer lugar usando

```
SELECT max(temp_min) FROM clima;
```

```
max
-----
46
(1 linha)
```

Se for desejado saber a cidade (ou cidades) onde esta temperatura ocorreu pode-se tentar usar

```
SELECT cidade FROM clima WHERE temp_min = max(temp_min);
```

*ERRADO*

mas não vai funcionar, porque a função de agregação max não pode ser usada na cláusula WHERE (Esta restrição existe porque a cláusula WHERE determina as linhas que vão passar para o estágio de agregação e, portanto, precisa ser avaliada antes das funções de agregação serem computadas). Entretanto, como é geralmente o caso, a consulta pode ser reformulada para obter o resultado pretendido, o que será feito por meio de uma *subconsulta*:

```
SELECT cidade FROM clima
WHERE temp_min = (SELECT max(temp_min) FROM clima);
```

```
cidade
-----
São Francisco
(1 linha)
```

Isto está correto porque a subconsulta é uma ação independente, que calcula sua agregação em separado do que está acontecendo na consulta externa.

As agregações também são muito úteis em combinação com a cláusula GROUP BY. Por exemplo, pode ser obtida a maior temperatura mínima observada em cada cidade usando

```
SELECT cidade, max(temp_min)
FROM clima
GROUP BY cidade;
```

cidade	max
Hayward	37
São Francisco	46

(2 linhas)

produzindo uma linha de saída para cada cidade. Cada resultado da agregação é computado sobre as linhas da tabela correspondendo a uma cidade. As linhas agrupadas podem ser filtradas utilizando a cláusula `HAVING`

```
SELECT cidade, max(temp_min)
  FROM clima
  GROUP BY cidade
  HAVING max(temp_min) < 40;
```

cidade	max
Hayward	37

(1 linha)

que mostra os mesmos resultados, mas apenas para as cidades que possuem todos os valores de `temp_min` abaixo de 40. Para concluir, se desejarmos somente as cidades com nome começando pela letra “s” podemos escrever:

```
SELECT cidade, max(temp_min)
  FROM clima
  WHERE cidade LIKE 'S%'❶
  GROUP BY cidade
  HAVING max(temp_min) < 40;
```

❶ O operador `LIKE` faz correspondência com padrão, sendo explicado na Seção 9.7.

É importante compreender a interação entre as agregações e as cláusulas `WHERE` e `HAVING` do SQL. A diferença fundamental entre `WHERE` e `HAVING` é esta: `WHERE` seleciona as linhas de entrada antes dos grupos e agregações serem computados (portanto, controla quais linhas irão para o computo da agregação), enquanto `HAVING` seleciona linhas de grupo após os grupos e agregações serem computados. Portanto, a cláusula `WHERE` não pode conter funções de agregação; não faz sentido tentar utilizar uma agregação para determinar quais linhas serão a entrada da agregação. Por outro lado, a cláusula `HAVING` sempre contém funções de agregação (A rigor, é permitido escrever uma cláusula `HAVING` que não possua agregação, mas é desperdício: A mesma condição poderia ser utilizada de forma mais eficiente no estágio do `WHERE`).<sup>7</sup>

No exemplo anterior, a restrição do nome da cidade pode ser aplicada na cláusula `WHERE`, porque não necessita de nenhuma agregação, sendo mais eficiente que colocar a restrição na cláusula `HAVING`, porque evita realizar os procedimentos de agrupamento e agregação em todas as linhas que não atendem a cláusula `WHERE`.

## 2.8. Atualizações

As linhas existentes podem ser atualizadas utilizando o comando `UPDATE`. Suponha que foi descoberto que as leituras de temperatura estão todas mais altas 2 graus após 28 de novembro de 1994. Os dados podem ser atualizados da seguinte maneira:

```
UPDATE clima
  SET temp_max = temp_max - 2, temp_min = temp_min - 2
  WHERE data > '1994-11-28';
```

Agora vejamos o novo estado dos dados:

```
SELECT * FROM clima;
```

cidade	temp_min	temp_max	prcp	data
São Francisco	46	50	0.25	1994-11-27
São Francisco	41	55	0	1994-11-29
Hayward	35	52		1994-11-29

(3 linhas)

## 2.9. Exclusões

As linhas podem ser removidas da tabela através do comando `DELETE`. Suponha que não estamos mais interessados nos registros do clima em Hayward. Então precisamos excluir estas linhas da tabela.

```
DELETE FROM clima WHERE cidade = 'Hayward';
```

Todos os registros de clima pertencentes a Hayward são removidos.

```
SELECT * FROM clima;
```

cidade	temp_min	temp_max	prcp	data
São Francisco	46	50	0.25	1994-11-27
São Francisco	41	55	0	1994-11-29

(2 linhas)

Deve-se tomar cuidado com comandos na forma:

```
DELETE FROM nome_da_tabela;
```

Sem uma qualificação, o comando `DELETE` remove *todas* as linhas da tabela, deixando-a vazia. O sistema não solicita confirmação antes de realizar esta operação!

## Notas

1. Os arquivos `basics.sql` (`./basics.sql`) e `advanced.sql` (`./advanced.sql`) foram traduzidos e colocados como links nesta tradução. Para usá-los basta salvar os arquivos em disco, abrir o interpretador de comandos, tornar o diretório onde os arquivos foram salvos o diretório corrente, executar na linha de comando `psql -s meu_bd` e usar o comando `\i basics.sql` ou `\i advanced.sql` para executar o arquivo, como mostrado neste capítulo. (N. do T.)
2. Use o comando `make --version` para saber se o `make` utilizado é o `make` do GNU. (N. do T.)
3. Embora o `SELECT *` seja útil para consultas rápidas, geralmente é considerado um estilo ruim para código em produção, uma vez que a adição de uma coluna à tabela mudaria os resultados.
4. Para o Oracle 9i as palavras chave `UNIQUE` e `DISTINCT` são sinônimos, podendo ser usada qualquer uma das duas no comando `SELECT`. (N. do T.)
5. Em alguns sistemas de banco de dados, incluindo as versões antigas do PostgreSQL, a implementação do `DISTINCT` ordena automaticamente as linhas e, por isso, o `ORDER BY` é redundante. Mas isto não é requerido pelo padrão SQL, e o PostgreSQL corrente não garante que `DISTINCT` faça com que as linhas sejam ordenadas.
6. Oracle 9i — Se for especificado o operador `DISTINCT` no comando `SELECT`, então a cláusula `ORDER BY` não pode fazer referência a colunas que não aparecem na lista de seleção. (Foi observado que esta restrição também se aplica ao PostgreSQL, ao DB2 e ao SQL Server). (N. do T.)
7. Consulte o Exemplo 7-1 (N. do T.)

# Capítulo 3. Funcionalidades avançadas

## 3.1. Introdução

Nos capítulos anteriores foi descrita a utilização básica da linguagem SQL para armazenar e acessar dados no PostgreSQL. Agora serão mostradas algumas funcionalidades mais avançadas da linguagem SQL que simplificam a gerência, e evitam a perda e a corrupção dos dados. No final serão vistas algumas extensões do PostgreSQL.

Em certas ocasiões este capítulo faz referência aos exemplos encontrados no Capítulo 2 para modificá-los ou melhorá-los, portanto recomenda-se que este capítulo já tenha sido lido. Alguns exemplos do presente capítulo também se encontram no arquivo `advanced.sql` (`./advanced.sql`) no diretório do tutorial. Este arquivo também contém dados dos exemplos a serem carregados, que não serão repetidos aqui (consulte a Seção 2.1 para saber como usar este arquivo).

## 3.2. Visões

Reveja as consultas na Seção 2.6. Supondo que a consulta combinando os registros de clima e de localização das cidades seja de particular interesse para um aplicativo, mas que não se deseja digitar esta consulta toda vez que for necessária, então é possível criar uma *visão* baseada na consulta, atribuindo um nome a esta consulta pelo qual será possível referenciá-la como se fosse uma tabela comum.

```
CREATE VIEW minha_visao AS
    SELECT cidade, temp_min, temp_max, prcp, data, localizacao
    FROM clima, cidades
    WHERE cidade = nome;
```

```
SELECT * FROM minha_visao;
```

Fazer livre uso de visões é um aspecto chave de um bom projeto de banco de dados SQL. As visões permitem encapsular, atrás de interfaces que não mudam, os detalhes da estrutura das tabelas, que podem mudar na medida em que os aplicativos evoluem.

As visões podem ser utilizadas em praticamente todos os lugares onde uma tabela real pode ser utilizada. Construir visões baseadas em visões não é raro.

## 3.3. Chaves estrangeiras

Reveja as tabelas `clima` e `cidades` no Capítulo 2. Considere o seguinte problema: Desejamos ter certeza que não serão inseridas linhas na tabela `clima` sem que haja um registro correspondente na tabela `cidades`. Isto é chamado de manter a *integridade referencial* dos dados. Em sistemas de banco de dados muito simples poderia ser implementado (caso fosse) olhando primeiro a tabela `cidades` para verificar se existe a linha correspondente e, depois, inserir ou rejeitar a nova linha de `clima`. Esta abordagem possui vários problemas, e é muito inconveniente, por isso o PostgreSQL pode realizar esta operação por você.

A nova declaração das tabelas ficaria assim:

```
CREATE TABLE cidades (
    cidade      varchar(80) primary key,
    localizacao point
);

CREATE TABLE clima (
    cidade      varchar(80) references cidades(cidade),
    temp_min    int,
    temp_max    int,
    prcp        real,
    data        date
);
```

Agora, ao se tentar inserir uma linha inválida:

```
INSERT INTO clima VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');
```

```
ERROR: insert or update on table "clima" violates foreign key constraint "clima_cidade_fkey"
```

```
DETAIL: Key (cidade)=(Berkeley) is not present in table "cidades".
```

```
-- Tradução da mensagem
```

```
ERRO: inserção ou atualização na tabela "clima" viola a restrição de chave estrangeira "clima_cidade_fkey"
```

```
DETALHE: Chave (cidade)=(Berkeley) não está presente na tabela "cidades".
```

O comportamento das chaves estrangeiras pode receber ajuste fino no aplicativo. Não iremos além deste exemplo simples neste tutorial, mas consulte o Capítulo 5 para obter informações adicionais. Com certeza o uso correto de chaves estrangeiras melhora a qualidade dos aplicativos de banco de dados, portanto incentivamos muito que se aprenda a usá-las.

### 3.4. Transações

*Transação* é um conceito fundamental de todo sistema de banco de dados. O ponto essencial da transação é englobar vários passos em uma única operação de tudo ou nada. Os estados intermediários entre os passos não são vistos pelas demais transações simultâneas e, se ocorrer alguma falha que impeça a transação chegar até o fim, então nenhum dos passos intermediários irá afetar o banco de dados de forma alguma.

Por exemplo, considere um banco de dados de uma instituição financeira contendo o saldo da conta corrente de vários clientes, assim como o saldo total dos depósitos de cada agência. Suponha que se deseje transferir \$100.00 da conta da Alice para a conta do Bob. Simplificando barbaramente, os comandos SQL para esta operação seriam:

```
UPDATE conta_corrente SET saldo = saldo - 100.00
```

```
WHERE nome = 'Alice';
```

```
UPDATE filiais SET saldo = saldo - 100.00
```

```
WHERE nome = (SELECT nome_filial FROM conta_corrente WHERE nome = 'Alice');
```

```
UPDATE conta_corrente SET saldo = saldo + 100.00
```

```
WHERE nome = 'Bob';
```

```
UPDATE filiais SET saldo = saldo + 100.00
```

```
WHERE nome = (SELECT nome_filial FROM conta_corrente WHERE nome = 'Bob');
```

Os detalhes destes comandos não são importantes aqui; o importante é o fato de existirem várias atualizações distintas envolvidas para realizar uma operação bem simples. A contabilidade quer ter certeza que todas as atualizações são realizadas, ou que nenhuma delas é realizada. Não é interessante uma falha no sistema fazer com que Bob receba \$100.00 que não foi debitado da Alice. Também a Alice não continuará sendo uma cliente satisfeita se o dinheiro for debitado da conta dela e não for creditado na de Bob. É necessário garantir que, caso aconteça algo errado no meio da operação, nenhum dos passos executados até este ponto produza efeito. Agrupar as atualizações em uma *transação* dá esta garantia. Uma transação é dita como sendo *atômica*: do ponto de vista das outras transações, ou a transação acontece completamente ou nada acontece.

Desejamos, também, ter a garantia de estando a transação completa e aceita pelo sistema de banco de dados, que esta fique permanentemente gravada, e não seja perdida mesmo no caso de acontecer uma pane logo em seguida. Por exemplo, se estiver sendo registrado saque em dinheiro pelo Bob não se deseja, de forma alguma, que o débito em sua conta corrente desapareça por causa de uma pane ocorrida logo depois dele sair da agência. Um banco de dados transacional garante que todas as atualizações realizadas por uma transação ficam registradas em meio de armazenamento permanente (ou seja, em disco), antes da transação ser considerada completa.

Outra propriedade importante dos bancos de dados transacionais está muito ligada à noção de atualizações atômicas: quando várias transações estão executando simultaneamente, cada uma delas não deve enxergar as alterações incompletas efetuadas pelas outras. Por exemplo, se uma transação está ocupada totalizando o saldo de todas as agências, não pode ser visto o débito efetuado na agência da Alice mas ainda não creditado na agência do Bob, nem o contrário. Portanto, as transações devem ser tudo ou nada não apenas em termos do efeito permanente no banco de dados, mas também em termos de visibilidade durante o processamento. As atualizações feitas por uma transação em andamento não podem ser vistas pelas outras transações enquanto não terminar, quando todas as atualizações se tornam visíveis ao mesmo tempo.

No PostgreSQL a transação é definida envolvendo os comandos SQL da transação pelos comandos BEGIN e COMMIT. Sendo assim, a nossa transação bancária ficaria:

```
BEGIN;
UPDATE conta_corrente SET saldo = saldo - 100.00
    WHERE nome = 'Alice';
-- etc etc
COMMIT;
```

Se no meio da transação for decidido que esta não deve ser efetivada (talvez porque tenha sido visto que o saldo da Alice ficou negativo), pode ser executado o comando `ROLLBACK` em vez do `COMMIT` para fazer com que todas as atualizações sejam canceladas.

O PostgreSQL, na verdade, trata todo comando SQL como sendo executado dentro de uma transação. Se não for utilizado o comando `BEGIN`, então cada comando possui um `BEGIN` e, se der tudo certo, um `COMMIT` individual envolvendo-o. Um grupo de comandos envolvidos por um `BEGIN` e um `COMMIT` é algumas vezes chamado de *bloco de transação*.

**Nota:** Algumas bibliotecas cliente emitem um comando `BEGIN` e um comando `COMMIT` automaticamente, fazendo com que seja obtido o efeito de um bloco de transação sem ser perguntado. Verifique a documentação da interface utilizada.

É possível controlar os comandos na transação de uma forma mais granular utilizando os *pontos de salvamento* (savepoints). Os pontos de salvamento permitem cancelar partes da transação seletivamente, e efetivar as demais partes. Após definir o ponto de salvamento, através da instrução `SAVEPOINT`, é possível cancelar a transação até o ponto de salvamento, se for necessário, usando `ROLLBACK TO`. Todas as alterações no banco de dados efetuadas entre a definição do ponto de salvamento e o cancelamento são desprezadas, mas as alterações efetuadas antes do ponto de salvamento são mantidas.

Após cancelar até o ponto de salvamento este ponto de salvamento continua definido e, portanto, é possível cancelar várias vezes. Ao contrário, havendo certeza que não vai ser mais necessário cancelar até o ponto de salvamento, o ponto de salvamento pode ser liberado, para que o sistema possa liberar alguns recursos. Deve-se ter em mente que liberar ou cancelar até um ponto de salvamento libera, automaticamente, todos os pontos de salvamento definidos após o mesmo.

Tudo isto acontece dentro do bloco de transação e, portanto, nada disso é visto pelas outras sessões do banco de dados. Quando o bloco de transação é efetivado, as ações efetivadas se tornam visíveis como uma unidade para as outras sessões, enquanto as ações canceladas nunca se tornam visíveis.

Recordando o banco de dados da instituição financeira, suponha que devesse ser debitado \$100.00 da conta da Alice e creditado na conta do Bob, mas que foi descoberto em seguida que era para ser creditado na conta do Wally. Isso poderia ser feito utilizando pontos de salvamento como mostrado abaixo:

```
BEGIN;
UPDATE conta_corrente SET saldo = saldo - 100.00
    WHERE nome = 'Alice';
SAVEPOINT meu_ponto_de_salvamento;
UPDATE conta_corrente SET saldo = saldo + 100.00
    WHERE nome = 'Bob';
-- uai ... o certo é na conta do Wally
ROLLBACK TO meu_ponto_de_salvamento;
UPDATE conta_corrente SET saldo = saldo + 100.00
    WHERE nome = 'Wally';
COMMIT;
```

Obviamente este exemplo está simplificado ao extremo, mas é possível efetuar um grau elevado de controle sobre a transação através do uso de pontos de salvamento. Além disso, a instrução `ROLLBACK TO` é a única forma de obter novamente o controle sobre um bloco de transação colocado no estado interrompido devido a um erro, fora cancelar completamente e começar tudo de novo.

### 3.5. Herança

Herança é um conceito de banco de dados orientado a objeto, que abre novas possibilidades interessantes ao projeto de banco de dados.

Vamos criar duas tabelas: a tabela *idades* e a tabela *capitais*. Como é natural, as capitais também são cidades e, portanto, deve existir alguma maneira para mostrar implicitamente as capitais quando todas as cidades são mostradas. Se formos bastante perspicazes, poderemos criar um esquema como este:

```

CREATE TABLE capitais (
    nome          text,
    populacao     real,
    altitude      int,    -- (em pés)
    estado        char(2)
);

CREATE TABLE interior (
    nome          text,
    populacao     real,
    altitude      int     -- (em pés)
);

CREATE VIEW cidades AS
    SELECT nome, populacao, altitude FROM capitais
    UNION
    SELECT nome, populacao, altitude FROM interior;

```

Este esquema funciona bem para as consultas, mas não é bom quando é necessário atualizar várias linhas, entre outras coisas.

Esta é uma solução melhor:

```

CREATE TABLE cidades (
    nome          text,
    populacao     real,
    altitude      int     -- (em pés)
);

CREATE TABLE capitais (
    estado        char(2)
) INHERITS (cidades);

```

Neste caso, as linhas da tabela *capitais* herdam todas as colunas (nome, populacao e altitude) da sua tabela ancestral *cidades*. O tipo da coluna nome é *text*, um tipo nativo do PostgreSQL para cadeias de caracteres de tamanho variável. As capitais dos estados possuem uma coluna a mais chamada estado, que armazena a sigla do estado. No PostgreSQL uma tabela pode herdar de nenhuma, uma, ou de várias tabelas.

Por exemplo, a consulta abaixo retorna os nomes de todas as cidades, incluindo as capitais dos estados, localizadas a uma altitude superior a 500 pés:

```

SELECT nome, altitude
FROM cidades
WHERE altitude > 500;

```

nome	altitude
Las Vegas	2174
Mariposa	1953
Madison	845

(3 linhas)

Por outro lado, a consulta abaixo traz todas as cidades que não são capitais de estado e estão situadas a uma altitude superior a 500 pés:

```

SELECT nome, altitude
FROM ONLY cidades
WHERE altitude > 500;

```

nome	altitude
Las Vegas	2174
Mariposa	1953

(2 linhas)

Nesta consulta a palavra chave `ONLY` antes de `idades` indica que a consulta deve ser efetuada apenas na tabela `idades`, sem incluir as tabelas abaixo de `idades` na hierarquia de herança. Muitos comandos mostrados até agora — `SELECT`, `UPDATE` e `DELETE` — permitem usar a notação `ONLY`.

**Nota:** Embora a hierarquia seja útil com frequência, como não está integrada às restrições de unicidade e de chave estrangeira, sua utilidade é limitada. Consulte a Seção 5.5 para obter mais detalhes.

### 3.6. Conclusão

O PostgreSQL possui muitas funcionalidades não abordadas neste tutorial introdutório, o qual está orientado para os usuários com pouca experiência na linguagem SQL. Estas funcionalidades são mostradas com mais detalhes no restante deste livro.

Se for necessário mais material introdutório, por favor visite o sítio do PostgreSQL na Web (<http://www.postgresql.org>) para obter indicações sobre onde encontrar este material.



## II. A linguagem SQL

Esta parte descreve a utilização da linguagem SQL no PostgreSQL. Começa descrevendo a sintaxe geral do SQL e, depois, explica como criar estruturas para armazenar dados, como carregar o banco de dados e como consultá-lo. A parte intermediária mostra os tipos de dado disponíveis e as funções utilizadas nos comandos SQL. O restante trata de vários aspectos importantes para ajustar o banco de dados para obter um desempenho otimizado.

As informações contidas nesta parte estão dispostas de maneira que um usuário inexperiente possa seguir do princípio ao fim para obter uma compreensão completa dos tópicos, sem ser necessário fazer referência a partes posteriores muitas vezes. A intenção foi criar capítulos auto-contidos, de modo que os usuários avançados possam ler os capítulos individualmente conforme haja necessidade. As informações nesta parte estão apresentadas sob forma de narrativa, sendo cada unidade um tópico. Os leitores à procura de uma descrição completa de um determinado comando devem consultar a Parte VI.

Os leitores desta parte devem saber como conectar ao banco de dados PostgreSQL e executar comandos SQL. Incentivamos os leitores não familiarizados com estes procedimentos lerem primeiro a Parte I. Normalmente os comandos SQL são executados utilizando o terminal interativo do PostgreSQL `psql`, mas outros programas com funcionalidades equivalentes também podem ser utilizados.

# Capítulo 4. Sintaxe da linguagem SQL

Este capítulo descreve a sintaxe <sup>1</sup> da linguagem SQL, estabelecendo a base para compreender os próximos capítulos que descrevem detalhadamente como os comandos SQL são utilizados para definir e modificar os dados.

Aconselha-se aos usuários já familiarizados com a linguagem SQL a leitura cuidadosa deste capítulo, porque existem várias regras e conceitos implementados pelos bancos de dados SQL de forma inconsistente, ou específicos do PostgreSQL.

## 4.1. Estrutura léxica

Uma entrada SQL é constituída por uma seqüência de *comandos*. Um comando é composto por uma seqüência de *termos* (tokens), terminada por um ponto-e-vírgula (“;”). O fim do fluxo de entrada também termina o comando. Quais termos são válidos depende da sintaxe particular de cada comando. <sup>2</sup>

Um termo pode ser uma *palavra chave*, um *identificador*, um *identificador entre aspas*, um *literal* (ou constante), ou um caractere especial. Geralmente os termos são separados por espaço em branco (espaço, tabulação ou nova-linha), mas não há necessidade se não houver ambigüidade (normalmente só acontece quando um caractere especial está adjacente a um termo de outro tipo).

Além disso, podem existir *comentários* na entrada SQL. Os comentários não são termos, na realidade são equivalentes a espaço em branco.

Abaixo está mostrada uma entrada SQL válida (sintaticamente) para servir de exemplo:

```
SELECT * FROM MINHA_TABELA;  
UPDATE MINHA_TABELA SET A = 5;  
INSERT INTO MINHA_TABELA VALUES (3, 'oi você');
```

Esta é uma seqüência de três comandos, um por linha (embora isto não seja requerido; pode haver mais de um comando na mesma linha, e um único comando pode ocupar várias linhas).

A sintaxe do SQL não é muito coerente em relação a quais termos identificam comandos e quais são operandos ou parâmetros. Geralmente os primeiros termos são o nome do comando e, portanto, no exemplo mostrado acima pode-se dizer que estão presentes os comandos “SELECT”, “UPDATE” e “INSERT”. Entretanto, para exemplificar, o comando UPDATE sempre requer que o termo SET apareça em uma determinada posição, e esta forma particular do comando INSERT também requer a presença do termo VALUES para estar completa. As regras precisas da sintaxe de cada comando estão descritas na Parte VI.

### 4.1.1. Identificadores e palavras chave

Os termos SELECT, UPDATE e VALUES mostrados no exemplo acima são exemplos de *palavras chave*, ou seja, palavras que possuem um significado definido na linguagem SQL. Os termos MINHA\_TABELA e A são exemplos de *identificadores*, os quais identificam nomes de tabelas, colunas e outros objetos do banco de dados, dependendo do comando onde são utilizados. Portanto, algumas vezes são simplesmente chamados de “nomes”. As palavras chave e os identificadores possuem a mesma estrutura léxica, significando que não é possível saber se o termo é um identificador ou uma palavra chave sem conhecer a linguagem. A relação completa das palavras chave pode ser encontrada no Apêndice C.

Os identificadores e as palavras chave do SQL devem iniciar por uma letra (a-z e, também, letras com diacrítico <sup>3</sup> - áéc... - e letras não latinas), ou o caractere sublinhado (\_). Os demais caracteres de um identificador, ou da palavra chave, podem ser letras, sublinhados, dígitos (0-9) ou o cifrão (\$). Deve ser observado que, de acordo com o padrão SQL, o cifrão não é permitido em identificadores e, portanto, pode tornar o aplicativo menos portátil. O padrão SQL não irá definir palavra chave contendo dígitos, ou começando ou terminando por sublinhado e, portanto, os identificadores com esta forma estão a salvo contra possíveis conflitos com extensões futuras do padrão.

O sistema não utiliza mais que NAMEDATALEN-1 caracteres de um identificador; podem ser escritos nomes mais longos nos comandos, mas são truncados. Por padrão NAMEDATALEN é 64 e, portanto, o comprimento máximo de um identificador é 63. Se este limite causar problema, pode ser aumentado modificando a constante NAMEDATALEN no arquivo src/include/postgres\_ext.h).

Os identificadores e as palavras chave não fazem distinção entre letras maiúsculas e minúsculas. Portanto,

```
UPDATE MINHA_TABELA SET A = 5;
```

pode ser escrito de forma equivalente como

```
uPDaTE minha_tabela SeT a = 5;
```

Normalmente utiliza-se a convenção de escrever as *palavras chave em letras maiúsculas* e os *nomes em letras minúsculas*, como mostrado abaixo:

```
UPDATE minha_tabela SET a = 5;
```

Existe um segundo tipo de identificador: o *identificador delimitado* ou *identificador entre aspas*, formado pela colocação de uma sequência arbitrária de caracteres entre aspas ("). Um identificador delimitado é sempre um identificador, e nunca uma palavra chave. Portanto, "select" pode ser usado para fazer referência a uma tabela ou coluna chamada "select", enquanto select sem aspas sempre é uma palavra chave ocasionando, por isso, um erro do analisador quando usado onde um nome de tabela ou de coluna for esperado. O exemplo acima pode ser reescrito utilizando identificadores entre aspas como mostrado abaixo:

```
UPDATE "minha_tabela" SET "a" = 5;
```

Identificadores entre aspas podem conter qualquer caractere que não seja a própria aspas (Para incluir uma aspas, devem ser escritas duas aspas). Esta funcionalidade permite criar nomes de tabelas e de colunas que não seriam possíveis de outra forma, como os contendo espaços ou e-comercial (&). O limite do comprimento ainda se aplica.

Colocar um identificador entre aspas torna diferente as letras maiúsculas e minúsculas, enquanto as letras dos nomes não envoltos por aspas são sempre convertidas em minúsculas. Por exemplo, os identificadores FOO, foo e "foo" são considerados o mesmo identificador pelo PostgreSQL, mas "FOO" e "foo" são diferentes dos três primeiros e entre si.

A transformação das letras dos nomes que não estão entre aspas em minúsculas feita pelo PostgreSQL é incompatível com o padrão SQL, que especifica a transformação em maiúsculas das letras dos nomes que não estão entre aspas. Portanto, foo deveria ser equivalente a "FOO", e não a "foo", de acordo com o padrão. Se for desejado desenvolver aplicativos portáteis, aconselha-se a colocar o nome *sempre* entre aspas, ou *nunca* entre aspas.

#### Exemplo 4-1. Utilização de letras acentuadas em nomes de tabelas

Este exemplo tem por finalidade mostrar a utilização de letras acentuadas nos nomes de tabelas. Deve ser observado o problema na conversão de letras maiúsculas e minúsculas acentuadas utilizando o idioma C.<sup>4</sup>

```
=> SELECT name, setting FROM pg_settings WHERE name LIKE 'lc%';
```

name	setting
lc_collate	C
lc_ctype	C
lc_messages	C
lc_monetary	C
lc_numeric	C
lc_time	C

(6 linhas)

```
=> CREATE TABLE AÇÃO(cod_ação int, nome_ação text);
```

```
=> \dt
```

Lista de relações			
Esquema	Nome	Tipo	Dono
public	aÇÃO	tabela	postgres
public	teste_abc	tabela	postgres
public	testeaabc	tabela	postgres

(3 linhas)

```
-- No exemplo acima ÇÃ não foi convertido em minúsculas
```

```
=> \dt AÇÃO
Não foi encontrada nenhuma relação correspondente.
=> \dt ação
Não foi encontrada nenhuma relação correspondente.
=> \dt "ação"
```

Lista de relações

Esquema	Nome	Tipo	Dono
public	ação	tabela	postgres

(1 linha)

```
-- Os exemplos acima mostram que só ação entre aspas corresponde ao nome da tabela.
-- Abaixo a tabela é criada com letras minúsculas.
=> CREATE TABLE ação(cod_ação int, nome_ação text);
=> \dt
```

Lista de relações

Esquema	Nome	Tipo	Dono
public	ação	tabela	postgres
public	teste_abc	tabela	postgres
public	testeabc	tabela	postgres

(3 linhas)

```
=> \dt ação
```

Lista de relações

Esquema	Nome	Tipo	Dono
public	ação	tabela	postgres

(1 linha)

```
=> \dt AÇÃO
```

Lista de relações

Esquema	Nome	Tipo	Dono
public	ação	tabela	postgres

(1 linha)

```
-- Nos exemplos acima foram bem-sucedidas a utilização tanto de ação quanto de AÇÃO.
=> INSERT INTO AÇÃO VALUES (1,'primeira ação');
ERRO: a relação "ação" não existe
=> INSERT INTO ação VALUES (1,'primeira ação');
INSERT 1665900 1
-- Nos exemplos acima só foi bem-sucedida a utilização de ação.
-- Para todas as letras do nome ficarem maiúsculas estes devem
-- ser escritos com letras maiúsculas e colocados entre aspas.
=> CREATE TABLE "AÇÃO"("COD_AÇÃO" int, "NOME_AÇÃO" text);
=> teste=# \dt
```

```

          Lista de relações
Esquema | Nome | Tipo | Dono
-----+-----+-----+-----
public  | AÇÃO | tabela | postgres
public  | teste_abc | tabela | postgres
public  | testeaabc | tabela | postgres
(3 linhas)

```

```
=> \dt "AÇÃO"
```

```

          Lista de relações
Esquema | Nome | Tipo | Dono
-----+-----+-----+-----
public  | AÇÃO | tabela | postgres
(1 linha)

```

```
=> INSERT INTO "AÇÃO" ("COD_AÇÃO", "NOME_AÇÃO") VALUES (1,'primeira ação');
```

```
=> SELECT * FROM "AÇÃO";
```

```

COD_AÇÃO | NOME_AÇÃO
-----+-----
1 | primeira ação
(1 linha)

```

### 4.1.2. Constantes

Existem três tipos de *constante com tipo implícito* no PostgreSQL: cadeias de caracteres, cadeias de bits e numéricas. As constantes também podem ser especificadas com tipo explícito, o que permite uma representação mais precisa, e um tratamento mais eficiente por parte do sistema. Estas alternativas são mostradas nas próximas subseções.

#### 4.1.2.1. Constantes do tipo cadeia de caracteres

Uma constante cadeia de caracteres no SQL é uma sequência arbitrária de caracteres envolta por apóstrofos (') como, por exemplo, 'Esta é uma cadeia de caracteres'. A forma de escrever um apóstrofo dentro de uma constante cadeia de caracteres, em conformidade com o padrão SQL, é colocar dois apóstrofos adjacentes como, por exemplo, 'Maria D'Almeida'. O PostgreSQL também permite utilizar a contrabarra ("") como caractere de escape para colocar apóstrofos dentro de cadeia de caracteres como, por exemplo, 'Maria D\'Almeida'.

Outra extensão do PostgreSQL é permitir a utilização dos escapes de contrabarra no estilo da linguagem C: \b para voltar apagando (backspace), \f para avanço de formulário (form feed), \n para nova-linha (newline), \r para retorno do carro (carriage return), \t para tabulação (tab) e \xxx, onde xxx é um número octal, é o byte com o código correspondente (É sua responsabilidade que as seqüências de byte criadas sejam caracteres válidos no conjunto de codificação de caracteres do servidor). Qualquer outro caractere vindo após a contrabarra é interpretado literalmente. Portanto, para incluir uma contrabarra em uma constante do tipo cadeia de caracteres devem ser escritas duas contrabarras adjacentes.

O caractere com o código zero não pode estar presente em uma constante cadeia de caracteres.

Duas constantes cadeia de caracteres separadas apenas por espaço em branco com *pelo menos um caractere de nova-linha*, são concatenadas e tratadas efetivamente como se a cadeia de caracteres tivesse sido escrita em uma constante. Por exemplo:

```
SELECT 'foo'
'bar';
```

equivale a

```
SELECT 'foobar';
```

mas

```
SELECT 'foo'      'bar';
```

não é uma sintaxe válida (este comportamento, um tanto ao quanto esquisito, é especificado no padrão SQL; o PostgreSQL está seguindo o padrão).

#### Exemplo 4-2. Constantes cadeia de caracteres ocupando mais de uma linha

Este exemplo tem por finalidade mostrar a utilização de uma constante cadeia de caracteres ocupando mais de uma linha para inserir dados em uma tabela. No Oracle e no DB2 há necessidade do operador de concatenação ||, enquanto no SQL Server há necessidade do operador de concatenação +. Só não houve necessidade do operador de concatenação no PostgreSQL.<sup>5</sup>

PostgreSQL 8.0.0:

```
=> CREATE TABLE "AÇÃO"("COD_AÇÃO" int, "NOME_AÇÃO" text);

=> INSERT INTO "AÇÃO" ("COD_AÇÃO", "NOME_AÇÃO") VALUES (1,'um nome'
(> ' de ação'
(> ' muito longo');

=> SELECT * FROM "AÇÃO";
```

```

COD_AÇÃO |          NOME_AÇÃO
-----+-----
1 | um nome de ação muito longo
(1 linha)
```

SQL Server 2000:

```
CREATE TABLE "AÇÃO"("COD_AÇÃO" int, "NOME_AÇÃO" text)

INSERT INTO "AÇÃO" ("COD_AÇÃO", "NOME_AÇÃO") VALUES (1,'um nome' +
' de ação' +
' muito longo')

SELECT * FROM "AÇÃO"
```

```

COD_AÇÃO    NOME_AÇÃO
-----
1          um nome de ação muito longo
(1 row(s) affected)
```

Oracle 10g:

```
SQL> CREATE TABLE "AÇÃO"("COD_AÇÃO" int, "NOME_AÇÃO" varchar2(32));

SQL> INSERT INTO "AÇÃO" ("COD_AÇÃO", "NOME_AÇÃO") VALUES (1,'um nome' ||
2  ' de ação' ||
3  ' muito longo');

SQL> SELECT * FROM "AÇÃO";
```

```

COD_AÇÃO NOME_AÇÃO
-----
1 um nome de ação muito longo
```

DB2 8.1:

```
DB2SQL92> CREATE TABLE "AÇÃO"("COD_AÇÃO" int, "NOME_AÇÃO" varchar(32));
DB2SQL92> INSERT INTO "AÇÃO" ("COD_AÇÃO", "NOME_AÇÃO") VALUES (1,'um nome' ||
DB2SQL92> ' de ação' ||
DB2SQL92> ' muito longo');
DB2SQL92> SELECT * FROM "AÇÃO";
```

```

COD_AÇÃO    NOME_AÇÃO
-----
1  um nome de ação muito longo
```

#### 4.1.2.2. Constantes cadeia de caracteres delimitadas por cifrão

Embora a sintaxe padrão para especificar constantes cadeia de caracteres seja muitas vezes conveniente, quando a cadeia de caracteres desejada contém vários apóstrofos ou contrabarras pode ser difícil compreendê-la, uma vez que estes devem ser duplicados. Para tornar o comando mais legível em uma situação como esta, o PostgreSQL disponibiliza uma outra maneira para escrever constantes cadeia de caracteres, chamada de “delimitação por cifrão” (*dollar quoting*). Uma constante cadeia de caracteres delimitada por cifrão é formada por um cifrão (\$), uma “marca” opcional com zero ou mais caracteres, outro cifrão, uma sequência arbitrária de caracteres constituindo o conteúdo da cadeia de caracteres, o cifrão, a mesma “marca” que iniciou esta delimitação por cifrão, e um cifrão. Para exemplificar são mostradas abaixo duas formas diferentes de especificar a cadeia de caracteres “Maria D’Almeida” usando delimitação por cifrão:

```
$$Maria D'Almeida$$
$UmaMarca$Maria D'Almeida$UmaMarca$
```

Deve ser observado que, dentro da cadeia de caracteres delimitada por cifrão, os apóstrofos podem ser utilizados sem necessidade de escape. Na verdade, nenhum caractere dentro de uma cadeia de caracteres delimitada por cifrão recebe escape: o conteúdo da cadeia de caracteres é sempre escrito literalmente. As contrabarras não são caracteres especiais, nem são os caracteres de cifrão, a menos que sejam parte da sequência correspondente a marca de abertura.

É possível aninhar constantes cadeias de caracteres delimitadas por cifrão escolhendo marcas diferentes a cada nível de aninhamento. É utilizado com mais frequência ao escrever definições de funções. Por exemplo:

```
$function$
BEGIN
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

Aqui a sequência `$q$[\t\r\n\v\\]$q$` representa o literal cadeia de caracteres delimitada por cifrão `[\t\r\n\v\\]`, que será reconhecido quando o corpo da função for executado pelo PostgreSQL. Mas uma vez que a sequência não corresponde ao delimitador de cifrão externo `$function$`, são apenas mais alguns caracteres dentro da constante no que diz respeito à cadeia de caracteres externa.

A marca de uma cadeia de caracteres delimitada por cifrão, se houver, segue as mesmas regras de um identificador não delimitado, exceto que não pode conter o caractere cifrão. Nas marcas, letras maiúsculas e minúsculas são diferentes e, portanto, `$marca$Conteúdo da cadeia de caracteres$marca$` está correto, mas `$MARCA$Conteúdo da cadeia de caracteres$marca$` não está.

Uma cadeia de caracteres delimitada por cifrão, vindo após uma palavra chave ou um identificador, deve ser separada do mesmo por um espaço em branco, senão o cifrão delimitador da cadeia de caracteres delimitada por cifrão será considerado como parte do identificador que o precede.

A cadeia de caracteres delimitada por cifrão não faz parte do padrão SQL mas é, muitas vezes, uma forma mais conveniente de escrever literais cadeias de caracteres complicados do que a forma em conformidade com o padrão usando apóstrofos. É particularmente útil para representar constantes cadeias de caracteres dentro de outras constantes, geralmente necessário na definição de funções em linguagens procedurais. Com a sintaxe de apóstrofos, cada contrabarra no exemplo acima deveria ser escrita como quatro contrabarras, que seriam reduzidas para duas contrabarras na análise da constante cadeia de caracteres original e, depois, para uma quando a constante cadeia de caracteres interna fosse re-analisada durante a execução da função.

#### 4.1.2.3. Constantes do tipo cadeia de bits

Uma constante do tipo cadeia de bits se parece com uma constante do tipo cadeia de caracteres contendo a letra B (maiúscula ou minúscula) imediatamente antes do apóstrofo de abertura (sem espaços separadores) como, por exemplo, `B'1001'`. Os únicos caracteres permitidos dentro de uma constante do tipo cadeia de bits são 0 e 1.

Como forma alternativa, constantes do tipo cadeia de bits podem ser especificadas usando a notação hexadecimal, colocando a letra X (maiúscula ou minúscula) no início como, por exemplo, `X'1FF'`. Esta notação equivale a uma constante do tipo cadeia de bits contendo quatro dígitos binários para cada dígito hexadecimal.

As duas formas de constantes do tipo cadeia de bits podem ocupar mais de uma linha, da mesma forma que uma constante do tipo cadeia de caracteres. A delimitação por cifrão não pode ser utilizada para o tipo cadeia de bits.

#### 4.1.2.4. Constantes numéricas

São aceitas constantes numéricas nas seguintes formas gerais:

```
dígitos
dígitos.[dígitos][e[+-]dígitos]
[dígitos].dígitos[e[+-]dígitos]
dígitose[+-]dígitos
```

onde *dígitos* são um ou mais dígitos decimais (0 a 9). Deve haver pelo menos um dígito antes ou depois do ponto decimal, se este for usado. Deve haver pelo menos um dígito após a marca de expoente (e), caso esteja presente. Não podem existir espaços ou outros caracteres incorporados à constante. Deve ser observado que os sinais menos e mais que antecedem a constante não são, na verdade, considerados parte da constante, e sim um operador aplicado à constante.

Abaixo são mostrados alguns exemplos de constantes numéricas válidas:

```
42
3.5
4.
.001
5e2
1.925e-3
```

Uma constante numérica não contendo o ponto decimal nem o expoente é presumida, inicialmente, como sendo do tipo `integer`, se o seu valor for apropriado para o tipo `integer` (32 bits); senão é presumida como sendo do tipo `bigint`, se o seu valor for apropriado para o tipo `bigint` (64 bits); caso contrário, é assumida como sendo do tipo `numeric`. As constantes que contêm pontos decimais e/ou expoentes são sempre presumidas inicialmente como sendo do tipo `numeric`.

O tipo de dado atribuído inicialmente para a constante numérica é apenas o ponto de partida para os algoritmos de resolução de tipo. Na maioria dos casos, a constante é automaticamente convertida no tipo mais apropriado conforme o contexto. Quando for necessário, pode-se impor que o valor numérico seja interpretado como sendo de um tipo de dado específico, definindo a conversão a ser aplicada. Por exemplo, pode-se impor que o valor numérico seja tratado como sendo do tipo `real` (`float4`) escrevendo:

```
REAL '1.23'  -- estilo cadeia de caracteres
1.23::REAL   -- estilo PostgreSQL (histórico)
```

Na verdade estes são apenas casos especiais da notação geral de conversão mostrada a seguir.

#### 4.1.2.5. Constantes de outros tipos

Pode ser declarada uma constante de um tipo *arbitrário* utilizando uma das seguintes notações:

```
tipo 'cadeia de caracteres'
'cadeia de caracteres::tipo'
CAST ( 'cadeia de caracteres' AS tipo )
```

O texto da constante cadeia de caracteres é passado para a rotina de conversão da entrada para o tipo chamado *tipo*. O resultado é uma constante do tipo indicado. A conversão explícita de tipo pode ser omitida caso não haja ambigüidade com relação ao tipo que a constante deva ter (por exemplo, quando é atribuída diretamente para uma coluna de uma tabela), neste caso é convertida automaticamente.

A constante cadeia de caracteres pode ser escrita utilizando tanto a notação regular do padrão SQL quanto a delimitação por cifrão.

Também é possível especificar a conversão de tipo utilizando a sintaxe semelhante à chamada de função

```
nome_do_tipo ( 'cadeia de caracteres' )
```

mas nem todos os nomes de tipo podem ser usados desta forma; consulte a Seção 4.2.8 para obter informações adicionais.

As sintaxes `::`, `CAST()` e chamada de função também podem ser utilizadas para especificar a conversão de tipo em tempo de execução para expressões arbitrárias, conforme mostrado na Seção 4.2.8. Porém, a forma `tipo 'cadeia de caracteres'` somente pode ser utilizada para especificar o tipo de uma constante literal. Outra restrição com relação à



sintaxe *tipo 'cadeia de caracteres'*, é que não funciona em tipo matriz (arrays); deve ser usado `::` ou `CAST()` para especificar o tipo de uma constante matriz.

### 4.1.3. Operadores

Um nome de operador é uma sequência com até `NAMEDATALEN-1` (por padrão 63) caracteres da seguinte lista:

`+ - * / < > = ~ ! @ # % ^ & | ` ?`

Entretanto, existem algumas poucas restrições para os nomes de operadores:

- Não podem ocorrer as seqüências `--` e `/*` em nenhuma posição no nome do operador, porque são consideradas início de comentário.
- Um nome de operador com vários caracteres não pode terminar por `+` ou por `-`, a não ser que o nome também contenha ao menos um dos seguintes caracteres: `~ ! @ # % ^ & | ` ?`. Por exemplo, `@-` é um nome de operador permitido, mas `*-` não é. Esta restrição permite ao PostgreSQL analisar comandos em conformidade com o padrão SQL sem requerer espaços entre os termos.

Ao trabalhar com nomes de operadores fora do padrão SQL, normalmente é necessário separar operadores adjacentes por espaço para evitar ambigüidade. Por exemplo, se for definido um operador unário-esquerdo chamado `@`, não poderá ser escrito `X*@Y`; deverá ser escrito `X* @Y`, para garantir que o PostgreSQL leia dois nomes de operadores e não apenas um.

### 4.1.4. Caracteres especiais

Alguns caracteres não alfanuméricos possuem significado especial diferente de ser um operador. Os detalhes da utilização podem ser encontrados nos locais onde a sintaxe do respectivo elemento é descrita. Esta seção se destina apenas a informar a existência e fazer um resumo das finalidades destes caracteres.

- O caractere cifrão (`$`) seguido por dígitos é utilizado para representar parâmetros posicionais no corpo da definição de uma função ou declaração preparada. Em outros contextos, o caractere cifrão pode ser parte de um identificador ou de uma constante cadeia de caracteres delimitada por cifrão.
- Os parênteses (`()`) possuem seu significado usual de agrupar expressões e impor a precedência. Em alguns casos, os parênteses são requeridos como parte da sintaxe fixada para um determinado comando SQL.
- Os colchetes (`[]`) são utilizados para selecionar elementos da matriz. Consulte a Seção 8.10 para obter mais informações sobre matrizes.
- As vírgulas (`,`) são utilizadas em algumas construções sintáticas para separar elementos da lista.
- O ponto-e-vírgula (`;`) termina um comando SQL, não podendo aparecer em nenhum lugar dentro do comando, exceto dentro de constantes do tipo cadeia de caracteres ou identificadores entre aspas.
- Os dois-pontos (`:`) são utilizados para selecionar “fatias” de matrizes (consulte a Seção 8.10). Em certos dialetos do SQL, como a linguagem SQL incorporada, os dois-pontos são utilizados como prefixo dos nomes das variáveis.
- O asterisco (`*`) é utilizado em alguns contextos para denotar todos os campos da linha de uma tabela ou de um valor composto. Também possui um significado especial quando utilizado como argumento da função de agregação `COUNT`.
- O ponto (`.`) é utilizado nas constantes numéricas, e para separar os nomes de esquemas, tabelas e colunas.

### 4.1.5. Comentários

Um comentário é uma seqüência arbitrária de caracteres começando por dois hífen e prosseguindo até o fim da linha como, por exemplo:

```
-- Este é um comentário em conformidade com o padrão SQL-92
```

Como alternativa, podem ser utilizados blocos de comentários no estilo C:

```
/* comentário de várias linhas
 * com aninhamento: /* bloco de comentário aninhado */
 */
```

onde o comentário começa por `/*` e se estende até encontrar a ocorrência correspondente de `*/`. Estes blocos de comentários podem estar aninhados, conforme especificado no padrão SQL, mas diferentemente da linguagem C, permitindo transformar em comentário grandes blocos de código contendo blocos de comentários.<sup>6 7</sup>

Os comentários são removidos do fluxo de entrada antes de prosseguir com a análise sintática, sendo substituídos por espaço em branco.

#### 4.1.6. Precedência léxica

A Tabela 4-1 mostra a precedência e a associatividade dos operadores no PostgreSQL. A maioria dos operadores possui a mesma precedência e associatividade esquerda. A precedência e a associatividade dos operadores está codificada no analisador, podendo ocasionar um comportamento contra-intuitivo; por exemplo, os operadores booleanos `<` e `>` possuem uma precedência diferente dos operadores booleanos `<=` e `>=`. Também, em alguns casos é necessário adicionar parênteses ao utilizar uma combinação de operadores unários e binários. Por exemplo,

```
SELECT 5 ! - 6;
```

será analisado como

```
SELECT 5 ! (- 6);
```

porque o analisador não possui a menor idéia — até ser tarde demais — que o `!` é definido como operador unário-direito (*postfix*), e não um operador binário colocado entre os operandos (*infix*). Neste caso, para obter o comportamento desejado deve ser escrito:

```
SELECT (5 !) - 6;
```

Este é o preço a ser pago pela extensibilidade.

**Tabela 4-1. Precedência dos operadores (decrecente)**

Operador/Elemento	Associatividade	Descrição
.	esquerda	separador de nome de tabela/coluna
::	esquerda	conversão de tipo estilo PostgreSQL
[ ]	esquerda	seleção de elemento de matriz
-	direita	menos unário
^	esquerda	exponenciação
* / %	esquerda	multiplicação, divisão, módulo
+ -	esquerda	adição, subtração
IS		IS TRUE, IS FALSE, IS UNKNOWN, IS NULL
ISNULL		teste de nulo
NOTNULL		teste de não nulo
(qualquer outro)	esquerda	os demais operadores nativos e os definidos pelo usuário
IN		membro de um conjunto
BETWEEN		contido em um intervalo
OVERLAPS		sobreposição de intervalo de tempo
LIKE ILIKE SIMILAR		correspondência de padrão em cadeia de caracteres
< >		menor que, maior que
=	direita	igualdade, atribuição

Operador/Elemento	Associatividade	Descrição
NOT	direita	negação lógica
AND	esquerda	conjunção lógica
OR	esquerda	disjunção lógica

Deve ser observado que as regras de precedência dos operadores também se aplicam aos operadores definidos pelos usuários que possuem os mesmos nomes dos operadores nativos mencionados acima. Por exemplo, se for definido pelo usuário um operador “+” para algum tipo de dado personalizado, este terá a mesma precedência do operador “+” nativo, não importando o que faça.

Quando um nome de operador qualificado pelo esquema é utilizado na sintaxe `OPERATOR` como, por exemplo, em

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

a construção `OPERATOR` é assumida como tendo a precedência padrão mostrada na Tabela 4-1 para “qualquer outro” operador. Isto é sempre verdade, não importando qual o nome do operador especificado dentro de `OPERATOR( )`.

## 4.2. Expressões de valor

As expressões de valor são utilizadas em diversos contextos, como na lista de seleção do comando `SELECT`, como novos valores das colunas nos comandos `INSERT` e `UPDATE`, e na condição de procura em vários comandos. Algumas vezes o resultado de uma expressão de valor é chamado de *escalar*, para distingui-lo do resultado de uma expressão de tabela (que é uma tabela). As expressões de valor são, portanto, chamadas também de *expressões escalares* (ou mesmo simplesmente de *expressões*). A sintaxe da expressão permite o cálculo de valores a partir de partes primitivas utilizando operações aritméticas, lógicas, de conjunto e outras.

A expressão de valor é uma das seguintes:

- Um valor constante ou literal.
- Uma referência a coluna.
- Uma referência a parâmetro posicional, no corpo da definição de função ou de comando preparado.
- Uma expressão de índice.
- Uma expressão de seleção de campo.
- Uma chamada de operador.
- Uma chamada de função.
- Uma expressão de agregação.
- Uma conversão de tipo.
- Uma subconsulta escalar.
- Um construtor de matriz.
- Um construtor de linha.
- Outra expressão de valor entre parênteses, útil para agrupar subexpressões e mudar precedências.

Em acréscimo a esta lista, existem diversas construções que podem ser classificadas como uma expressão, mas que não seguem qualquer regra geral de sintaxe. Possuem, normalmente, a semântica de uma função ou de um operador, sendo explicadas no local apropriado no Capítulo 9. Um exemplo é a cláusula `IS NULL`.

As constantes já foram mostradas na Seção 4.1.2. As próximas seções discutem as demais opções.

### 4.2.1. Referências a coluna

Uma coluna pode ser referenciada usando a forma

```
correlação.nome_da_coluna
```

onde *correlação* é o nome de uma tabela (possivelmente qualificado pelo nome do esquema), ou um aliás para a tabela definido por meio da cláusula `FROM`, ou uma das palavras chave `NEW` ou `OLD` (`NEW` e `OLD` somente podem aparecer nas regras de reescrita, enquanto os outros nomes de correlação podem ser usados em qualquer declaração SQL). O nome da correlação e o ponto separador podem ser omitidos, se o nome da coluna for único entre todas as tabelas utilizadas no comando corrente. (Consulte também o Capítulo 7.)

#### 4.2.2. Parâmetros posicionais

É utilizada uma referência a um parâmetro posicional para indicar um valor fornecido externamente a uma declaração SQL. Os parâmetros são utilizados na definição de funções SQL e de comandos preparados. Algumas bibliotecas cliente também suportam a especificação de valores de dados separado da cadeia de caracteres do comando SQL e, nestes casos, os parâmetros são utilizados para fazer referência a valores de dados fora de linha. A forma de fazer referência a um parâmetro é:

`$number`

Por exemplo, considere a definição da função `dept` como sendo:

```
CREATE FUNCTION dept(text) RETURNS dept
  AS $$ SELECT * FROM dept WHERE nome = $1 $$
LANGUAGE SQL;
```

Neste caso, `$1` será substituído pelo primeiro argumento da função quando esta for chamada.

#### 4.2.3. Índices

Se uma expressão produzir um valor do tipo matriz, então um elemento específico do valor matricial pode ser extraído escrevendo:

`expressão[índice]`

e vários elementos adjacentes (uma “fatia da matriz”) pode ser extraída escrevendo:

`expressão[índice_inferior:índice_superior]`

(Neste caso, os colchetes `[ ]` devem aparecer literalmente). Cada *índice* é por si só uma expressão, que deve produzir um valor inteiro.

Geralmente a *expressão* matricial deve estar entre parênteses, mas os parênteses podem ser omitidos quando a expressão a ser indexada é apenas a referência a uma coluna ou um parâmetro posicional. Podem ser concatenados vários índices quando a matriz original for multidimensional. Por exemplo:

```
minha_tabela.matriz_coluna[4]
minha_tabela.matriz_duas_dim[17][34]
$1[10:42]
(funcao_matriz(a,b))[42]
```

No último exemplo os parênteses são requeridos. Consulte a Seção 8.10 para obter informações adicionais sobre matrizes.

#### 4.2.4. Escolha de campo

Se uma expressão produzir um valor do tipo composto (tipo linha), então pode-se extrair um campo específico da linha escrevendo:

`expressão.nome_do_campo`

Geralmente a *expressão* de linha deve estar entre parênteses, mas os parênteses podem ser omitidos quando a expressão de seleção for apenas uma referência a tabela ou um parâmetro posicional. Por exemplo,

```
minha_tabela.minha_coluna
$1.alguma_coluna
(funcao_de_linha(a,b)).col3
```

(Portanto, uma referência a coluna qualificada é, na verdade, apenas um caso especial da sintaxe de seleção de campo).

#### 4.2.5. Chamadas de operador

Existem três sintaxes possíveis para chamada de operador:

*expressão operador expressão* (operador binário-intermediário)  
*operador expressão* (operador unário-esquerdo)  
*expressão operador* (operador unário-direito)

onde o termo *operador* segue as regras de sintaxe da Seção 4.1.3, ou é uma das palavras chave AND, OR ou NOT, ou é um nome de operador qualificado na forma:

`OPERATOR(esquema.nome_do_operador)`

Quais são os operadores existentes, e se são unários ou binários, depende de quais operadores foram definidos pelo sistema e pelo usuário. O Capítulo 9 descreve os operadores nativos.<sup>8 9 10</sup>

#### 4.2.6. Chamadas de função

A sintaxe para chamada de função é o nome da função (possivelmente qualificado pelo nome do esquema), seguido por sua lista de argumentos entre parênteses:

*função* ([*expressão* [, *expressão* ... ]])

Por exemplo, a função abaixo calcula a raiz quadrada de 2:

`sqrt(2)`

A lista de funções nativas está no Capítulo 9. Podem ser adicionadas outras funções pelo usuário.

#### 4.2.7. Funções de agregação

Uma *expressão de agregação* representa a aplicação de uma função de agregação nas linhas selecionadas pela consulta. Uma função de agregação reduz vários valores de entrada a um único valor de saída, tal como a soma ou a média dos valores entrados. A sintaxe da expressão de agregação é uma das seguintes:

*nome\_da\_agregação* (*expressão*)  
*nome\_da\_agregação* (ALL *expressão*)  
*nome\_da\_agregação* (DISTINCT *expressão*)  
*nome\_da\_agregação* ( \* )

onde *nome\_da\_agregação* é uma agregação definida anteriormente (possivelmente qualificado pelo nome do esquema), e *expressão* é qualquer expressão de valor que não contenha uma expressão de agregação.

A primeira forma de expressão de agregação chama a função de agregação para todas as linhas de entrada onde a expressão fornecida produz um valor não nulo (na verdade, é decisão da função de agregação ignorar ou não os valores nulos — porém, todas as funções padrão o fazem). A segunda forma é idêntica à primeira, porque ALL é o padrão. A terceira forma chama a função de agregação para todos os valores distintos não nulos da expressão, encontrados nas linhas de entrada. A última forma chama a função de agregação uma vez para cada linha de entrada independentemente do valor ser nulo ou não; como nenhum valor específico de entrada é especificado, geralmente é útil apenas para a função de agregação `count()`.

Por exemplo, `count(*)` retorna o número total de linhas de entrada; `count(f1)` retorna o número de linhas de entrada onde `f1` não é nulo; `count(distinct f1)` retorna o número de valores distintos não nulos de `f1`.

As funções de agregação predefinidas estão descritas na Seção 9.15. Podem ser adicionadas pelo usuário outras funções de agregação.

Uma expressão de agregação pode aparecer apenas na lista de resultados ou na cláusula HAVING do comando SELECT. Seu uso é proibido nas outras cláusulas, tal como WHERE, porque estas cláusulas são avaliadas logicamente antes dos resultados das agregações estarem formados.

Quando uma expressão de agregação aparece em uma subconsulta (consulte a Seção 4.2.9 e a Seção 9.16), normalmente a agregação é avaliada a partir das linhas da subconsulta. Porém ocorre uma exceção quando o argumento da agregação

contém apenas variáveis do nível externo: a agregação então pertence ao nível externo mais próximo, sendo avaliada a partir das linhas desta consulta. A expressão de agregação como um todo é, então, uma referência externa para a subconsulta onde aparece, agindo como uma constante em qualquer avaliação da subconsulta. A restrição de aparecer apenas na lista de resultados ou na cláusula `HAVING` se aplica com respeito ao nível da consulta que a agregação pertence.

#### 4.2.8. Conversões de tipo

Uma conversão de tipo (`type cast`) especifica a conversão de um tipo de dado em outro. O PostgreSQL aceita duas sintaxes equivalentes para conversão de tipo:

```
CAST ( expressão AS tipo )
expressão::tipo
```

A sintaxe `CAST` está em conformidade com o padrão SQL; a sintaxe `::` é uma utilização histórica do PostgreSQL.

Quando a conversão é aplicada a uma expressão de valor de tipo conhecido, representa uma conversão em tempo de execução. A conversão será bem sucedida apenas se estiver disponível uma operação de conversão de tipo adequada. Deve ser observado que isto é sutilmente diferente da utilização de conversão com constantes, conforme mostrado na Seção 4.1.2.5. Uma conversão aplicada a um literal cadeia de caracteres sem adornos representa a atribuição inicial do tipo ao valor constante literal<sup>11 12</sup> e, portanto, será bem-sucedida para qualquer tipo (se o conteúdo do literal cadeia de caracteres possuir uma sintaxe válida para servir de entrada para o tipo de dado).

Geralmente a conversão explícita de tipo pode ser omitida quando não há ambigüidade em relação ao tipo que a expressão de valor deve produzir (por exemplo, quando é atribuída a uma coluna de tabela); o sistema aplica automaticamente a conversão de tipo nestes casos. Entretanto, a conversão automática de tipo é feita apenas para as conversões marcadas nos catálogos do sistema como “OK para aplicar implicitamente”. As outras conversões devem ser chamadas por meio da sintaxe de conversão explícita. Esta restrição tem por finalidade impedir que aconteçam conversões surpreendentes aplicadas em silêncio.

Também é possível especificar uma conversão de tipo utilizando a sintaxe na forma de função:

```
nome_do_tipo ( expressão )
```

Entretanto, somente funciona para os tipos cujos nomes também são válidos como nome de função. Por exemplo, `double precision` não pode ser utilizado desta maneira, mas a forma equivalente `float8` pode. Também, os nomes `interval`, `time` e `timestamp` somente podem ser utilizados desta maneira se estiverem entre aspas, devido a conflitos sintáticos. Portanto, o uso da sintaxe de conversão na forma de função pode ocasionar inconsistências, devendo ser evitada em novos aplicativos. (A sintaxe tipo chamada de função é, de fato, apenas uma chamada de função. Quando é utilizada uma das duas sintaxes padrão de conversão para fazer conversão em tempo de execução, internamente chama a função registrada para realizar esta conversão. Por convenção, estas funções de conversão possuem o mesmo nome de seu tipo de dado de saída e, portanto, a “sintaxe tipo função” não é nada mais do que a chamada direta à função de conversão subjacente. Como é óbvio, isto não é algo que um aplicativo portátil possa depender).

#### 4.2.9. Subconsultas escalares

Uma subconsulta escalar é um comando `SELECT` comum, entre parênteses, que retorna exatamente uma linha com uma coluna (consulte o Capítulo 7 para obter informações sobre como escrever consultas). O comando `SELECT` é executado e o único valor retornado é utilizado na expressão de valor envoltória. É errado utilizar uma consulta que retorne mais de uma linha ou mais de uma coluna como subconsulta escalar (porém, se durante uma determinada execução a subconsulta não retornar nenhuma linha, não acontece nenhum erro: o resultado escalar é assumido como nulo). A subconsulta pode fazer referência a variáveis da consulta envoltória, as quais atuam como constantes durante a avaliação da subconsulta. Veja, também, outras expressões envolvendo subconsultas na Seção 9.16.

Por exemplo, a consulta abaixo retorna a maior população de cidade de cada estado:

```
SELECT nome, (SELECT max(populacao) FROM cidades WHERE cidades.estado = estados.nome)
FROM estados;
```

#### 4.2.10. Construtores de matriz

Um construtor de matriz é uma expressão que constrói um valor matriz a partir dos valores de seus elementos membros. Um construtor de matriz simples é composto pela palavra chave `ARRAY`, um abre colchetes `[`, uma ou mais expressões (separadas por vírgula) para os valores dos elementos da matriz e, finalmente, um fecha colchetes `]`. Por exemplo,

```
SELECT ARRAY[1,2,3+4];
```

```

      array
-----
 {1,2,7}
(1 linha)
```

O tipo de dado do elemento da matriz é o tipo comum das expressões membro, determinado utilizando as mesmas regras das construções `UNION` e `CASE` (consulte a Seção 10.5).

Os valores matriz multidimensional podem ser construídos aninhando construtores de matriz. Nos construtores internos, a palavra chave `ARRAY` pode ser omitida. Por exemplo, estes dois comandos produzem o mesmo resultado:

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
```

```

      array
-----
 {{1,2},{3,4}}
(1 linha)
```

```
SELECT ARRAY[[1,2],[3,4]];
```

```

      array
-----
 {{1,2},{3,4}}
(1 linha)
```

Uma vez que as matrizes multidimensionais devem ser retangulares, os construtores internos no mesmo nível devem produzir submatrizes com dimensões idênticas.

Os elementos construtores de matriz multidimensional podem ser qualquer coisa que produza uma matriz do tipo apropriado, e não apenas uma construção sub-`ARRAY`. Por exemplo:

```
CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[1,2],[3,4], ARRAY[5,6],[7,8]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
```

```

      array
-----
 {{{1,2},{3,4}},{5,6},{7,8}},{9,10},{11,12}}}
(1 linha)
```

Também é possível construir uma matriz a partir do resultado de uma subconsulta. Nesta forma, o construtor de matriz é escrito com a palavra chave `ARRAY` seguida por uma subconsulta entre parênteses, e não entre colchetes. Por exemplo:

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
```

```

      ?column?
-----
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31}
(1 linha)
```

A subconsulta deve retornar uma única coluna. A matriz unidimensional produzida terá um elemento para cada linha no resultado da subconsulta, com o tipo do elemento correspondendo ao da coluna de saída da subconsulta.

O índice de um valor da matriz construído com ARRAY sempre começa por um. Para obter informações adicionais sobre matrizes consulte a Seção 8.10.

#### 4.2.11. Construtores de linha

Um construtor de linha é uma expressão que constrói um valor linha (também chamado de valor composto) a partir de valores de seus campos membros. Um construtor de linha é formado pela palavra chave ROW, um abre parênteses, zero ou mais expressões (separadas por vírgula) para os valores dos campos da linha e, finalmente, por um fecha parênteses. Por exemplo:

```
SELECT ROW(1,2.5,'isto é um teste');
```

A palavra chave ROW é opcional quando existe mais de uma expressão na lista.

Por padrão, o valor criado através da expressão ROW é de um tipo de registro anônimo. Se for necessário, pode ser convertido para um tipo composto com nome — tanto o tipo de linha de uma tabela, ou um tipo composto criado pelo comando CREATE TYPE AS. Pode ser necessária uma conversão explícita para evitar ambigüidade. Por exemplo:

```
CREATE TABLE minha_tabela(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(minha_tabela) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- Não é necessária nenhuma conversão, porque só existe uma getf1()
SELECT getf1(ROW(1,2.5,'isto é um teste'));

    getf1
-----
      1
(1 linha)

CREATE TYPE meu_tipo_de_linha AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(meu_tipo_de_linha) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- Agora é necessária uma conversão para indicar a função a ser chamada:
SELECT getf1(ROW(1,2.5,'isto é um teste'));
ERRO:  a função getf1(record) não é única

SELECT getf1(ROW(1,2.5,'isto é um teste')::minha_tabela);

    getf1
-----
      1
(1 linha)

SELECT getf1(CAST(ROW(11,'isto é um teste',2.5) AS meu_tipo_de_linha));

    getf1
-----
     11
(1 linha)
```

Os construtores de linha podem ser utilizados para construir valores compostos a serem armazenados em colunas de tabelas de tipo composto, ou serem passados para funções que recebem parâmetros compostos. Também é possível comparar dois valores linha ou testar uma linha com IS NULL ou IS NOT NULL como, por exemplo:

```
SELECT ROW(1,2.5,'isto é um teste') = ROW(1, 3, 'não é o mesmo');

SELECT ROW(a, b, c) IS NOT NULL FROM tabela;
```

Para obter informações adicionais consulte a Seção 9.17. Os construtores de linha também podem ser utilizados conectados a subconsultas conforme mostrado na Seção 9.16.



### 4.2.12. Regras para avaliação de expressão

A ordem de avaliação das subexpressões não é definida. Em particular, as entradas de um operador ou função não são necessariamente avaliadas da esquerda para a direita, ou em qualquer outra ordem fixada.

Além disso, se o resultado da expressão puder ser determinado avaliando apenas algumas de suas partes, então as outras subexpressões podem nem ser avaliadas. Por exemplo, se for escrito

```
SELECT true OR alguma_funcao();
```

então `alguma_funcao()` não será (provavelmente) chamada. Este é o mesmo caso de quando é escrito

```
SELECT alguma_funcao() OR true;
```

Deve ser observado que isto não é o mesmo que os “curtos circuitos” esquerda para direita de operadores booleanos encontrados em algumas linguagens de programação.

Como consequência, não é bom utilizar funções com efeitos colaterais como parte de expressões complexas. É particularmente perigoso confiar em efeitos colaterais, ou na ordem de avaliação nas cláusulas `WHERE` e `HAVING`, porque estas cláusulas são extensamente reprocessadas como parte do desenvolvimento do plano de execução. As expressões booleanas (combinações de `AND/OR/NOT`) nestas cláusulas podem ser reorganizadas em qualquer forma permitida pelas leis da álgebra booleana.

Quando for essencial obrigar a ordem de avaliação, pode ser utilizada uma construção `CASE` (consulte a Seção 9.13). Por exemplo, esta é uma forma não confiável para tentar evitar uma divisão por zero na cláusula `WHERE`:

```
SELECT ... WHERE x <> 0 AND y/x > 1.5;
```

Mas esta forma é segura:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

A construção `CASE` utilizada desta forma impede as tentativas de otimização devendo, portanto, ser utilizada apenas quando for necessário (Neste exemplo em particular, sem dúvida seria melhor evitar o problema escrevendo `y > 1.5*x`).

## Notas

1. *sintaxe* — do Lat. *syntaxe* < Gr. *śyntaxis*, arranjo, disposição — parte da estrutura gramatical de uma língua que contém as regras relativas à combinação das palavras em unidades maiores (como as orações), e as relações existentes entre as palavras dentro dessas unidades. PRIBERAM - Língua Portuguesa On-Line (<http://www.priberam.pt/dlpo/dlpo.aspx>). (N. do T.)
2. *léxico* — do Gr. *lěxicon*, relativo às palavras — dicionário de línguas clássicas antigas; dicionário abreviado; conjunto dos vocábulos de uma língua; dicionário dos vocábulos usados num domínio especializado (ciência, técnica). PRIBERAM - Língua Portuguesa On-Line (<http://www.priberam.pt/dlpo/dlpo.aspx>). (N. do T.)
3. *diacrítico* — do Gr. *diakritikós*, que se pode distinguir — diz-se dos sinais gráficos com que se notam os caracteres alfabéticos para lhe dar um valor especial. PRIBERAM - Língua Portuguesa On-Line (<http://www.priberam.pt/dlpo/dlpo.aspx>). (N. do T.)
4. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
5. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
6. Oracle 9i — O manual não cita comentários aninhados, mas experiências com o SQL\*Plus mostraram que não é possível aninhar comentários `/*` dentro de comentários `/*`, mas que é possível colocar comentários `--` dentro de comentários `/*`. Comments ([http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/sql\\_elements7a.htm](http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/sql_elements7a.htm)). (N. do T.)
7. SQL Server 2000 — O Books Online só cita comentários `--` dentro de comentários `/*`, mas experiências com o Query Analyzer mostraram que é possível aninhar comentários `/*` dentro de comentários `/*`, mas a cor do comentário não fica correta. (N. do T.)

8. `binary infix operator` — foi traduzido como “operador binário-intermediário”, mas poderia ter sido traduzido como “operador binário infix”, onde infix significa “afixo no meio de uma palavra”, mas é um termo pouco conhecido. PRIBERAM - Língua Portuguesa On-Line (<http://www.priberam.pt/dlpo/dlpo.aspx>). (N. do T.)
9. `unary prefix operator` — foi traduzido como “operador unário-esquerdo”. (N. do T.)
10. `unary postfix operator` — foi traduzido como “operador unário-direito”. (N. do T.)
11. Oracle — Os termos literal e valor constante são sinônimos e referem a um valor de dado fixo. Por exemplo, 'JACK', 'BLUE ISLAND' e '101' são todos literais caractere; 5001 é um literal numérico. Literais caractere são envoltos por apóstrofos, o que permite ao Oracle distingui-los dos nomes dos objetos do esquema. Oracle9i SQL Reference - Literals ([http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/sql\\_elements3a.htm](http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/sql_elements3a.htm)) (N. do T.)
12. Cada tipo em Java possui “literais”, que são a maneira como os valores constantes daquele tipo são escritos. Ken Arnold e James Gosling - Programando em Java - Makron Books - 1997 (N. do T.)

# Capítulo 5. Definição de dados

Este capítulo mostra como criar as estruturas de banco de dados que armazenam os dados. Nos bancos de dados relacionais os dados são armazenados em tabelas, por isso a maior parte deste capítulo dedica-se a explicar como as tabelas são criadas e modificadas, e as funcionalidades disponíveis para controlar que dados podem ser armazenados nas tabelas. Em seguida é mostrado como as tabelas podem ser organizadas em esquemas, e como atribuir privilégios às tabelas. No final são vistas, superficialmente, outras funcionalidades que afetam o armazenamento dos dados, como visões, funções e gatilhos.

## 5.1. Noções básicas de tabela

Uma tabela em um banco de dados relacional é muito semelhante a uma tabela no papel: é formada por linhas e colunas. O número e a ordem das colunas são fixos, e cada coluna possui um nome. O número de linhas é variável, refletindo a quantidade de dados armazenados em um determinado instante. O padrão SQL não dá nenhuma garantia sobre a ordem das linhas na tabela. Quando a tabela é lida, as linhas aparecem em uma ordem aleatória, a não ser que a classificação seja requisitada explicitamente. Esta parte é descrita no Capítulo 7. Além disso, o SQL não atribui identificadores únicos para as linhas e, portanto, é possível existirem várias linhas totalmente idênticas na tabela. Isto é uma consequência do modelo matemático subjacente ao SQL, mas geralmente não é desejável. Mais adiante neste capítulo será mostrado como lidar com esta questão.

Cada coluna possui um tipo de dado. O tipo de dado restringe o conjunto de valores que podem ser atribuídos à coluna e atribui semântica<sup>1</sup> aos dados armazenados na coluna, de forma que estes possam ser processados. Por exemplo, uma coluna declarada como sendo de um tipo numérico não aceita cadeias de caracteres com texto arbitrário, e os dados armazenados nesta coluna podem ser utilizados para efetuar cálculos matemáticos. Ao contrário, uma coluna declarada como sendo do tipo cadeia de caracteres aceita praticamente qualquer espécie de dado, mas não pode ser usada para efetuar cálculos matemáticos, embora possam ser efetuadas outras operações, como a concatenação de cadeias de caracteres.

O PostgreSQL possui um extenso conjunto de tipos de dado nativos, adequados para muitos aplicativos. Os usuários também podem definir seus próprios tipos de dado. A maioria dos tipos de dado nativos possui nome e semântica óbvia, portanto uma explicação detalhada será postergada até o Capítulo 8. Alguns dos tipos de dado mais utilizados são o `integer` para números inteiros, `numeric` para números possivelmente fracionários, `text` para cadeias de caracteres, `date` para datas, `time` para valores da hora do dia, e `timestamp` para valores contendo tanto data quanto hora.

Para criar uma tabela utiliza-se o comando `CREATE TABLE`. Neste comando são especificados, ao menos, o nome da nova tabela, os nomes das colunas, e o tipo de dado de cada coluna. Por exemplo:

```
CREATE TABLE minha_primeira_tabela (  
    primeira_coluna text,  
    segunda_coluna integer  
);
```

Este comando cria a tabela chamada `minha_primeira_tabela` contendo duas colunas. A primeira coluna chama-se `primeira_coluna`, e possui o tipo de dado `text`; a segunda coluna chama-se `segunda_coluna`, e possui o tipo de dado `integer`. O nome da tabela e das colunas obedecem a sintaxe para identificadores explicada na Seção 4.1.1. Normalmente os nomes dos tipos também são identificadores, mas existem algumas exceções. Deve ser observado que a lista de colunas é envolta por parênteses, e os elementos da lista são separados por vírgula.

Obviamente, o exemplo anterior é muito artificial. Normalmente são dados nomes para as tabelas e para as colunas condizentes com as informações armazenadas. Sendo assim, vejamos um exemplo mais próximo da realidade:

```
CREATE TABLE produtos (  
    cod_prod    integer,  
    nome        text,  
    preco       numeric  
);
```

(O tipo `numeric` pode armazenar a parte fracionária, comum em valores monetários)

**Dica:** Quando são criadas tabelas inter-relacionadas, é aconselhável escolher um padrão coerente para atribuir nomes às tabelas e colunas. Por exemplo, existe a possibilidade de utilizar nomes de tabelas no singular ou no plural, e cada uma destas possibilidades é defendida por um teórico ou por outro.

Existe um limite de quantas colunas uma tabela pode conter. Dependendo dos tipos das colunas, pode ser entre 250 e 1600. Entretanto, definir uma tabela com esta quantidade de colunas é muito raro e, geralmente, torna o projeto questionável.

Se uma tabela não for mais necessária, pode-se removê-la utilizando o comando `DROP TABLE`. Por exemplo:

```
DROP TABLE minha_primeira_tabela;
DROP TABLE produtos;
```

Tentar remover uma tabela não existente é um erro. Entretanto, é comum os arquivos de script SQL tentarem remover a tabela incondicionalmente antes de criá-la, ignorando a mensagem de erro.

Se for necessário modificar uma tabela existente consulte a Seção 5.6 mais adiante neste capítulo.

Utilizando as ferramentas mostradas até este ponto é possível criar tabelas totalmente funcionais. O restante deste capítulo está relacionado com a adição de funcionalidades na definição da tabela para garantir a integridade dos dados, a segurança, ou a comodidade. Se você está ansioso para colocar dados nas tabelas neste instante, então pode ir direto para o Capítulo 6 e ler o restante deste capítulo depois.

## 5.2. Valor padrão

Pode ser atribuído um valor padrão a uma coluna. Quando é criada uma nova linha, e não é especificado nenhum valor para algumas de suas colunas, estas colunas são preenchidas com o valor padrão de cada uma delas. Além disso, um comando de manipulação de dados pode requerer explicitamente que a coluna receba o seu valor padrão, sem saber qual é este valor (os detalhes sobre os comandos de manipulação de dados estão no Capítulo 6).

Se não for declarado explicitamente nenhum valor padrão, o valor nulo será o valor padrão. Isto geralmente faz sentido, porque o valor nulo pode ser considerado como representando um dado desconhecido.

Na definição da tabela, o valor padrão é posicionado após o tipo de dado da coluna. Por exemplo:

```
CREATE TABLE produtos (
    cod_prod    integer PRIMARY KEY,
    nome        text,
    preco       numeric DEFAULT 9.99
);
```

O valor padrão pode ser uma expressão, avaliada sempre que for inserido o valor padrão (e *não* quando a tabela é criada). Um exemplo comum é uma coluna do tipo `timestamp` com o valor padrão `now()`, para que receba a data e hora de inserção da linha. Outro exemplo comum é a geração de um “número serial” para cada linha. No PostgreSQL isto é feito tipicamente através de algo como:

```
CREATE TABLE produtos (
    cod_prod integer DEFAULT nextval('produtos_cod_prod_seq'),
    ...
);
```

onde a função `nextval()` fornece valores sucessivos do *objeto de seqüência* (consulte a Seção 9.12). Esta situação é tão comum que existe uma forma abreviada da mesma:

```
CREATE TABLE produtos (
    cod_prod SERIAL,
    ...
);
```

A forma abreviada `SERIAL` é mostrada posteriormente na Seção 8.1.4.

## 5.3. Restrições

Os tipos de dado são uma forma de limitar os dados que podem ser armazenados na tabela. Entretanto, para muitos aplicativos a restrição obtida não possui o refinamento necessário. Por exemplo, uma coluna contendo preços de produtos provavelmente só pode aceitar valores positivos, mas não existe nenhum tipo de dado que aceite apenas números positivos. Um outro problema é que pode ser necessário restringir os dados de uma coluna com relação a outras colunas ou linhas. Por exemplo, em uma tabela contendo informações sobre produtos deve haver apenas uma linha para cada código de produto.

Para esta finalidade, a linguagem SQL permite definir restrições em colunas e tabelas. As restrições permitem o nível de controle sobre os dados da tabela que for desejado. Se o usuário tentar armazenar dados em uma coluna da tabela violando a restrição, ocasiona erro. Isto se aplica até quando o erro é originado pela definição do valor padrão.

### 5.3.1. Restrições de verificação

Uma restrição de verificação é o tipo mais genérico de restrição. Permite especificar que os valores de uma determinada coluna devem estar de acordo com uma expressão booleana (valor-verdade <sup>2</sup>). Por exemplo, para permitir apenas preços com valores positivos utiliza-se:

```
CREATE TABLE produtos (
    cod_prod    integer,
    nome        text,
    preco       numeric CHECK (preco > 0)
);
```

Como pode ser observado, a definição da restrição vem após o tipo de dado, assim como a definição do valor padrão. O valor padrão e a restrição podem estar em qualquer ordem. A restrição de verificação é formada pela palavra chave **CHECK** seguida por uma expressão entre parênteses. A expressão da restrição de verificação deve envolver a coluna sendo restringida, senão não fará muito sentido.

Também pode ser atribuído um nome individual para a restrição. Isto torna mais clara a mensagem de erro, e permite fazer referência à restrição quando se desejar alterá-la. A sintaxe é:

```
CREATE TABLE produtos (
    cod_prod    integer,
    nome        text,
    preco       numeric CONSTRAINT chk_preco_positivo CHECK (preco > 0)
);
```

Portanto, para especificar o nome da restrição deve ser utilizada a palavra chave **CONSTRAINT**, seguida por um identificador, seguido por sua vez pela definição da restrição (Se não for escolhido o nome da restrição desta maneira, o sistema escolhe um nome para a restrição).

Uma restrição de verificação também pode referenciar várias colunas. Supondo que serão armazenados o preço normal e o preço com desconto, e que se deseje garantir que o preço com desconto seja menor que o preço normal:

```
CREATE TABLE produtos (
    cod_prod          integer,
    nome              text,
    preco             numeric CHECK (preco > 0),
    preco_com_desconto numeric CHECK (preco_com_desconto > 0),
    CHECK (preco > preco_com_desconto)
);
```

As duas primeiras formas de restrição já devem ser familiares. A terceira utiliza uma nova sintaxe, e não está anexada a uma coluna em particular. Em vez disso, aparece como um item à parte na lista de colunas separadas por vírgula. As definições das colunas e as definições destas restrições podem estar em qualquer ordem.

Dizemos que as duas primeiras restrições são restrições de coluna, enquanto a terceira é uma restrição de tabela, porque está escrita separado das definições de colunas. As restrições de coluna também podem ser escritas como restrições de tabela, enquanto o contrário nem sempre é possível, porque supostamente a restrição de coluna somente faz referência à coluna em que está anexada (O PostgreSQL não impõe esta regra, mas deve-se segui-la se for desejado que a definição da tabela sirva para outros sistemas de banco de dados). O exemplo acima também pode ser escrito do seguinte modo:

```
CREATE TABLE produtos (
    cod_prod      integer,
    nome          text,
    preco         numeric,
    CHECK (preco > 0),
    preco_com_desconto numeric,
    CHECK (preco_com_desconto > 0),
    CHECK (preco > preco_com_desconto)
);
```

ou ainda

```
CREATE TABLE produtos (
    cod_prod      integer,
    nome          text,
    preco         numeric CHECK (preco > 0),
    preco_com_desconto numeric,
    CHECK (preco_com_desconto > 0 AND preco > preco_com_desconto)
);
```

É uma questão de gosto.

Podem ser atribuídos nomes para as restrições de tabela da mesma maneira que para as restrições de coluna:

```
CREATE TABLE produtos (
    cod_prod      integer,
    nome          text,
    preco         numeric,
    CHECK (preco > 0),
    preco_com_desconto numeric,
    CHECK (preco_com_desconto > 0),
    CONSTRAINT chk_desconto_valido CHECK (preco > preco_com_desconto)
);
```

Deve ser observado que a restrição de verificação está satisfeita se o resultado da expressão de verificação for verdade ou o valor nulo. Como a maioria das expressões retorna o valor nulo quando um dos operandos é nulo, estas expressões não impedem a presença de valores nulos nas colunas com restrição. Para garantir que a coluna não contém o valor nulo, deve ser utilizada a restrição de não nulo descrita a seguir.

### 5.3.2. Restrições de não-nulo

Uma restrição de não-nulo simplesmente especifica que uma coluna não pode assumir o valor nulo. Um exemplo da sintaxe:

```
CREATE TABLE produtos (
    cod_prod      integer NOT NULL,
    nome          text    NOT NULL,
    preco         numeric
);
```

A restrição de não-nulo é sempre escrita como restrição de coluna. A restrição de não-nulo é funcionalmente equivalente a criar uma restrição de verificação `CHECK (nome_da_coluna IS NOT NULL)`, mas no PostgreSQL a criação de uma restrição de não-nulo explícita é mais eficiente. A desvantagem é que não pode ser dado um nome explícito para uma restrição de não nulo criada deste modo.

Obviamente, uma coluna pode possuir mais de uma restrição, bastando apenas escrever uma restrição em seguida da outra:

```
CREATE TABLE produtos (
    cod_prod      integer NOT NULL,
    nome          text    NOT NULL,
    preco         numeric NOT NULL CHECK (preco > 0)
);
```

A ordem das restrições não importa, porque não determina, necessariamente, a ordem de verificação das restrições.

A restrição `NOT NULL` possui uma inversa: a restrição `NULL`. Isto não significa que a coluna deva ser nula, o que com certeza não tem utilidade. Em vez disto é simplesmente definido o comportamento padrão dizendo que a coluna pode ser nula. A restrição `NULL` não é definida no padrão SQL, não devendo ser utilizada em aplicativos portáteis (somente foi adicionada ao PostgreSQL para torná-lo compatível com outros sistemas de banco de dados). Porém, alguns usuários gostam porque torna fácil inverter a restrição no script de comandos. Por exemplo, é possível começar com

```
CREATE TABLE produtos (
    cod_prod    integer NULL,
    nome        text     NULL,
    preco       numeric NULL
);
```

e depois colocar a palavra chave `NOT` onde se desejar.

**Dica:** Na maioria dos projetos de banco de dados, a maioria das colunas deve ser especificada como não-nula.

### 5.3.3. Restrições de unicidade

A restrição de unicidade garante que os dados contidos na coluna, ou no grupo de colunas, é único em relação a todas as outras linhas da tabela. A sintaxe é

```
CREATE TABLE produtos (
    cod_prod    integer UNIQUE,
    nome        text,
    preco       numeric
);
```

quando escrita como restrição de coluna, e

```
CREATE TABLE produtos (
    cod_prod    integer,
    nome        text,
    preco       numeric,
    UNIQUE (cod_prod)
);
```

quando escrita como restrição de tabela.

Se uma restrição de unicidade faz referência a um grupo de colunas, as colunas são listadas separadas por vírgula:

```
CREATE TABLE exemplo (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

Isto especifica que a combinação dos valores das colunas indicadas deve ser único para toda a tabela, embora não seja necessário que cada uma das colunas seja única (o que geralmente não é).

Também é possível atribuir nomes às restrições de unicidade:

```
CREATE TABLE produtos (
    cod_prod    integer CONSTRAINT unq_cod_prod UNIQUE,
    nome        text,
    preco       numeric
);
```

De um modo geral, uma restrição de unicidade é violada quando existem duas ou mais linhas na tabela onde os valores de todas as colunas incluídas na restrição são iguais. Entretanto, os valores nulos não são considerados iguais nesta comparação. Isto significa que, mesmo na presença da restrição de unicidade, é possível armazenar um número ilimitado de linhas que contenham o valor nulo em pelo menos uma das colunas da restrição. Este comportamento está em

conformidade com o padrão SQL, mas já ouvimos dizer que outros bancos de dados SQL não seguem esta regra. Portanto, seja cauteloso ao desenvolver aplicativos onde se pretenda haver portabilidade.<sup>3 4 5</sup>

### 5.3.4. Chaves primárias

Tecnicamente a restrição de chave primária é simplesmente a combinação da restrição de unicidade com a restrição de não-nulo. Portanto, as duas definições de tabela abaixo aceitam os mesmos dados:

```
CREATE TABLE produtos (
    cod_prod    integer UNIQUE NOT NULL,
    nome        text,
    preco       numeric
);
```

```
CREATE TABLE produtos (
    cod_prod    integer PRIMARY KEY,
    nome        text,
    preco       numeric
);
```

As chaves primárias também podem restringir mais de uma coluna; a sintaxe é semelhante à da restrição de unicidade:

```
CREATE TABLE exemplo (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);
```

A chave primária indica que a coluna, ou grupo de colunas, pode ser utilizada como identificador único das linhas da tabela (Isto é uma consequência direta da definição da chave primária. Deve ser observado que a restrição de unicidade não fornece, por si só, um identificador único, porque não exclui os valores nulos). A chave primária é útil tanto para fins de documentação quanto para os aplicativos cliente. Por exemplo, um aplicativo contendo uma Interface de Usuário Gráfica (GUI), que permite modificar os valores das linhas, provavelmente necessita conhecer a chave primária da tabela para poder identificar as linhas de forma única.

Uma tabela pode ter no máximo uma chave primária (embora possa ter muitas restrições de unicidade e de não-nulo). A teoria de banco de dados relacional dita que toda tabela deve ter uma chave primária. Esta regra não é imposta pelo PostgreSQL, mas normalmente é melhor segui-la.

### 5.3.5. Chaves Estrangeiras

A restrição de chave estrangeira especifica que o valor da coluna (ou grupo de colunas) deve corresponder a algum valor existente em uma linha de outra tabela. Diz-se que a chave estrangeira mantém a *integridade referencial* entre duas tabelas relacionadas.

Supondo que já temos a tabela de produtos utilizada diversas vezes anteriormente:

```
CREATE TABLE produtos (
    cod_prod    integer PRIMARY KEY,
    nome        text,
    preco       numeric
);
```

Agora vamos assumir a existência de uma tabela armazenando os pedidos destes produtos. Desejamos garantir que a tabela de pedidos contenha somente pedidos de produtos que realmente existem. Para isso é definida uma restrição de chave estrangeira na tabela de pedidos fazendo referência à tabela de produtos:

```
CREATE TABLE pedidos (
    cod_pedido  integer PRIMARY KEY,
    cod_prod    integer REFERENCES produtos (cod_prod),
    quantidade  integer
);
```



Isto torna impossível criar um pedido com `cod_prod` não existente na tabela de produtos.

Nesta situação é dito que a tabela de pedidos é a tabela *que faz referência*, e a tabela de produtos é a tabela *referenciada*. Da mesma forma existem colunas fazendo referência e sendo referenciadas.

O comando acima pode ser abreviado escrevendo

```
CREATE TABLE pedidos (
    cod_pedido integer PRIMARY KEY,
    cod_prod   integer REFERENCES produtos,
    quantidade integer
);
```

porque, na ausência da lista de colunas, a chave primária da tabela referenciada é usada como a coluna referenciada.

A chave estrangeira também pode restringir e referenciar um grupo de colunas. Como usual, é necessário ser escrito na forma de restrição de tabela. Abaixo está mostrado um exemplo artificial da sintaxe:

```
CREATE TABLE t1 (
    a integer PRIMARY KEY,
    b integer,
    c integer,
    FOREIGN KEY (b, c) REFERENCES outra_tabela (c1, c2)
);
```

Obviamente, o número e tipo das colunas na restrição devem corresponder ao número e tipo das colunas referenciadas.

Pode ser atribuído um nome à restrição de chave estrangeira da forma habitual.

Uma tabela pode conter mais de uma restrição de chave estrangeira, o que é utilizado para implementar relacionamentos muitos-para-muitos entre tabelas. Digamos que existam as tabelas de produtos e de pedidos, e desejamos permitir que um pedido possa conter vários produtos (o que não é permitido na estrutura anterior). Podemos, então, utilizar a seguinte estrutura de tabela:

```
CREATE TABLE produtos (
    cod_prod   integer PRIMARY KEY,
    nome       text,
    preco      numeric
);

CREATE TABLE pedidos (
    cod_pedido integer PRIMARY KEY,
    endereco_entrega text,
    ...
);

CREATE TABLE itens_pedidos (
    cod_prod   integer REFERENCES produtos,
    cod_pedido integer REFERENCES pedidos,
    quantidade integer,
    PRIMARY KEY (cod_prod, cod_pedido)
);
```

Deve ser observado, também, que a chave primária está sobreposta às chaves estrangeiras na última tabela.

Sabemos que a chave estrangeira não permite a criação de pedidos não relacionados com algum produto. Porém, o que acontece se um produto for removido após a criação de um pedido fazendo referência a este produto? A linguagem SQL permite tratar esta situação também. Intuitivamente temos algumas opções:

- Não permitir a exclusão de um produto referenciado
- Excluir o pedido também
- Algo mais?

Para ilustrar esta situação, vamos implementar a seguinte política no exemplo de relacionamento muitos-para-muitos acima: Quando se desejar remover um produto referenciado por um pedido (através de `itens_pedidos`), isto não será permitido. Se um pedido for removido, os itens do pedido também serão removidos.

```
CREATE TABLE produtos (
    cod_prod    integer PRIMARY KEY,
    nome        text,
    preco       numeric
);

CREATE TABLE pedidos (
    cod_pedido   integer PRIMARY KEY,
    endereco_entrega text,
    ...
);

CREATE TABLE itens_pedidos (
    cod_prod     integer REFERENCES produtos ON DELETE RESTRICT,
    cod_pedido   integer REFERENCES pedidos  ON DELETE CASCADE,
    quantidade   integer,
    PRIMARY KEY (cod_prod, cod_pedido)
);
```

As duas opções mais comuns são restringir, ou excluir em cascata. `RESTRICT` não permite excluir a linha referenciada. `NO ACTION` significa que, se as linhas referenciadas ainda existirem quando a restrição for verificada, será gerado um erro; este é o comportamento padrão se nada for especificado (A diferença essencial entre estas duas opções é que `NO ACTION` permite postergar a verificação para mais tarde na transação, enquanto `RESTRICT` não permite). `CASCADE` especifica que, quando a linha referenciada é excluída, as linhas que fazem referência também devem ser excluídas automaticamente. Existem outras duas opções: `SET NULL` e `SET DEFAULT`. Estas opções fazem com que as colunas que fazem referência sejam definidas como nulo ou com o valor padrão, respectivamente, quando a linha referenciada é excluída. Deve ser observado que isto não evita a observância das restrições. Por exemplo, se uma ação especificar `SET DEFAULT`, mas o valor padrão não satisfizer a chave estrangeira, a operação não será bem-sucedida.

Semelhante a `ON DELETE` existe também `ON UPDATE`, chamada quando uma coluna referenciada é alterada (atualizada). As ações possíveis são as mesmas.

Mais informações sobre atualização e exclusão de dados podem ser encontradas no Capítulo 6.

Para terminar, devemos mencionar que a chave estrangeira deve referenciar colunas de uma chave primária ou de uma restrição de unicidade. Se a chave estrangeira fizer referência a uma restrição de unicidade, existem algumas possibilidades adicionais sobre como os valores nulos são correspondidos. Esta parte está explicada na documentação de referência para *CREATE TABLE*.

### 5.3.6. Exemplos do tradutor

#### Exemplo 5-1. Restrição de unicidade com valor nulo em chave única simples

Abaixo são mostrados exemplos de inserção de linhas contendo valor nulo no campo da chave única simples da restrição de unicidade. Deve ser observado que, nestes exemplos, o PostgreSQL e o Oracle consideram os valores nulos diferentes.

PostgreSQL 8.0.0:

```
=> \pset null '(nulo)'
=> CREATE TABLE tbl_unique (c1 int UNIQUE);
=> INSERT INTO tbl_unique VALUES (1);
=> INSERT INTO tbl_unique VALUES (NULL);
=> INSERT INTO tbl_unique VALUES (NULL);
=> INSERT INTO tbl_unique VALUES (2);
=> SELECT * FROM tbl_unique;
```

```

c1
-----
1
(nulo)
(nulo)
2
(4 linhas)

```

SQL Server 2000:

```

CREATE TABLE tbl_unique (c1 int UNIQUE)
INSERT INTO tbl_unique VALUES (1)
INSERT INTO tbl_unique VALUES (NULL)
INSERT INTO tbl_unique VALUES (NULL)
Violation of UNIQUE KEY constraint 'UQ__tbl_unique__37A5467C'.
Cannot insert duplicate key in object 'tbl_unique'.
The statement has been terminated.
INSERT INTO tbl_unique VALUES (2)
SELECT * FROM tbl_unique

```

```

c1
-----
NULL
1
2
(3 row(s) affected)

```

Oracle 10g:

```

SQL> SET NULL (nulo)
SQL> CREATE TABLE tbl_unique (c1 int UNIQUE);
SQL> INSERT INTO tbl_unique VALUES (1);
SQL> INSERT INTO tbl_unique VALUES (NULL);
SQL> INSERT INTO tbl_unique VALUES (NULL);
SQL> INSERT INTO tbl_unique VALUES (2);
SQL> SELECT * FROM tbl_unique;

```

```

c1
-----
1
(nulo)
(nulo)
2

```

### Exemplo 5-2. Restrição de unicidade com valor nulo em chave única composta

Abaixo são mostrados exemplos de inserção de linhas contendo valores nulos em campos da chave única composta da restrição de unicidade. Deve ser observado que, nestes exemplos, somente o PostgreSQL considera os valores nulos diferentes.

PostgreSQL 8.0.0:

```

=> \pset null '(nulo)'
=> CREATE TABLE tbl_unique (c1 int, c2 int, UNIQUE (c1, c2));
=> INSERT INTO tbl_unique VALUES (1,1);
=> INSERT INTO tbl_unique VALUES (1,NULL);
=> INSERT INTO tbl_unique VALUES (NULL,1);
=> INSERT INTO tbl_unique VALUES (NULL,NULL);
=> INSERT INTO tbl_unique VALUES (1,NULL);
=> SELECT * FROM tbl_unique;

```

```

c1  |  c2
-----+-----
1  |      1
1  | (nulo)

```

```
(nulo) |      1
(nulo) | (nulo)
      1 | (nulo)
(5 linhas)
```

SQL Server 2000:

```
CREATE TABLE tbl_unique (c1 int, c2 int, UNIQUE (c1, c2))
INSERT INTO tbl_unique VALUES (1,1)
INSERT INTO tbl_unique VALUES (1,NULL)
INSERT INTO tbl_unique VALUES (NULL,1)
INSERT INTO tbl_unique VALUES (NULL,NULL)
INSERT INTO tbl_unique VALUES (1,NULL)
Violation of UNIQUE KEY constraint 'UQ__tbl_unique__33D4B598'.
Cannot insert duplicate key in object 'tbl_unique'.
The statement has been terminated.
SELECT * FROM tbl_unique
```

```
c1          c2
-----
NULL        NULL
NULL        1
1           NULL
1           1
(4 row(s) affected)
```

Oracle 10g:

```
SQL> SET NULL (nulo)
SQL> CREATE TABLE tbl_unique (c1 int, c2 int, UNIQUE (c1, c2));
SQL> INSERT INTO tbl_unique VALUES (1,1);
SQL> INSERT INTO tbl_unique VALUES (1,NULL);
SQL> INSERT INTO tbl_unique VALUES (NULL,1);
SQL> INSERT INTO tbl_unique VALUES (NULL,NULL);
SQL> INSERT INTO tbl_unique VALUES (1,NULL);
INSERT INTO tbl_unique VALUES (1,NULL)
*
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.SYS_C005273) violated
SQL> SELECT * FROM tbl_unique;
```

```
      C1          C2
-----
      1           1
      1 (nulo)
(nulo)           1
(nulo)      (nulo)
```

### Exemplo 5-3. Cadeia de caracteres vazia e valor nulo

Abaixo são mostrados exemplos de consulta a uma tabela contendo tanto o valor nulo quanto uma cadeia de caracteres vazia em uma coluna. Deve ser observado que apenas o Oracle 10g não faz distinção entre a cadeia de caracteres vazia e o valor nulo. Foram utilizados os seguintes comandos para criar e inserir dados na tabela em todos os gerenciadores de banco de dados:

```
CREATE TABLE c (c1 varchar(6), c2 varchar(6));
INSERT INTO c VALUES ('x', 'x');
INSERT INTO c VALUES ('VAZIA', '');
INSERT INTO c VALUES ('NULA', null);
```

PostgreSQL 8.0.0:

```
=> \pset null '(nulo)'
=> SELECT * FROM c WHERE c2 IS NULL;
```

```

c1 | c2
-----+-----
NULA | (nulo)
(1 linha)

```

SQL Server 2000:

```
SELECT * FROM c WHERE c2 IS NULL
```

```

c1      c2
-----
NULA    NULL
(1 row(s) affected)

```

Oracle 10g:

```

SQL> SET NULL (nulo)
SQL> SELECT * FROM c WHERE c2 IS NULL;

```

```

C1      C2
-----
VAZIA   (nulo)
NULA    (nulo)

```

DB2 8.1:

```
DB2SQL92> SELECT * FROM c WHERE c2 IS NULL;
```

```

C1      C2
-----
NULA    -

```

#### Exemplo 5-4. Coluna sem restrição de não nulo em chave primária

Abaixo são mostrados exemplos de criação de uma tabela definindo uma chave primária em uma coluna que não é definida como não aceitando o valor nulo. O padrão SQL diz que, neste caso, a restrição de não nulo é implícita, mas o DB2 não implementa desta forma, enquanto o PostgreSQL, o SQL Server e o Oracle seguem o padrão. Também são mostrados comandos para exibir a estrutura da tabela nestes gerenciadores de banco de dados.

PostgreSQL 8.0.0:

```

=> CREATE TABLE c (c1 int, PRIMARY KEY(c1));
=> \d c

```

```

Tabela "public.c"
Coluna | Tipo | Modificadores
-----+-----
c1      | integer | not null

```

Índices:

```
"c_pkey" chave primária, btree (c1)
```

SQL Server 2000:

```

CREATE TABLE c (c1 int, PRIMARY KEY(c1));
sp_help c

```

```

Name Owner Type          Created_datetime
----
c      dbo    user table 2005-03-28 11:00:24.027

```

```

Column_name Type Computed Length Prec Scale Nullable
-----
c1          int   no         4      10    0      no

```

```

index_name      index_description      index_keys
-----
PK__c__403A8C7D clustered, unique, primary key located on PRIMARY c1

```

Oracle 10g:

```
SQL> CREATE TABLE c (c1 int, PRIMARY KEY(c1));
SQL> DESCRIBE c

Name Null?      Type
-----
C1      NOT NULL NUMBER(38)

SQL> SELECT index_name, index_type, tablespace_name, uniqueness
2 FROM user_indexes
3 WHERE table_name='C';

INDEX_NAME  INDEX_TYPE  TABLESPACE_NAME  UNIQUENES
-----
SYS_C005276  NORMAL      USERS              UNIQUE
```

DB2 8.1:

```
db2 => CREATE TABLE c (c1 int, PRIMARY KEY(c1))

SQL0542N  "C1" não pode ser uma coluna de uma chave primária ou exclusiva,
porque pode conter valores nulos.  SQLSTATE=42831

db2 => CREATE TABLE c (c1 int NOT NULL, PRIMARY KEY(c1))

DB20000I  O comando SQL terminou com sucesso.

db2 => DESCRIBE TABLE c

Nome da Tipo de  Nome do
coluna  esquema  tipo  Tamanho Escala Nulos
-----
C1      SYSIBM   INTEGER      4      0 Não

1 registro(s) selecionado(s).

db2 => DESCRIBE INDEXES FOR TABLE c SHOW DETAIL

Esquema do Nome do  Regra  Número de
índice  índice  exclusiva  colunas  Nomes de coluna
-----
SYSIBM   SQL050328184542990 P          1 +C1

1 registro(s) selecionado(s).
```

## 5.4. Colunas do sistema

Toda tabela possui diversas *colunas do sistema*, as quais são implicitamente definidas pelo sistema. Portanto, estes nomes não podem ser utilizados como nomes de colunas definidas pelo usuário (observe que esta restrição é diferente do nome ser uma palavra chave ou não; colocar o nome entre aspas não faz esta restrição deixar de ser aplicada). Não há necessidade dos usuários se preocuparem com estas colunas, basta apenas saber que elas existem.

`oid`

O identificador de objeto (`object ID`) de uma linha. É um número serial adicionado pelo PostgreSQL, automaticamente, a todas as linhas da tabela (a não ser que a tabela seja criada com `WITHOUT OIDS` e, neste caso, esta coluna não estará presente). O tipo desta coluna é `oid` (o mesmo nome da coluna); consulte a Seção 8.12 para obter informações adicionais sobre o tipo.

`tableoid`

O OID da tabela que contém esta linha. Este atributo é particularmente útil nas consultas fazendo seleção em hierarquias de herança, porque sem este atributo é difícil saber de que tabela se origina cada linha. Pode ser feita uma junção entre `tableoid` e a coluna `oid` de `pg_class` para obter o nome da tabela.

`xmin`

O identificador da transação de inserção (`transaction ID`) para esta versão da linha (Uma versão da linha é um estado individual da linha; cada atualização da linha cria uma nova versão de linha para a mesma linha lógica).

`cmin`

O identificador do comando, começando por zero, dentro da transação de inserção.

`xmax`

O identificador da transação de exclusão (`transaction ID`), ou zero para uma versão de linha não excluída. É possível que esta coluna seja diferente de zero em uma versão de linha visível: normalmente isto indica que a transação fazendo a exclusão ainda não foi efetivada (`commit`), ou que uma tentativa de exclusão foi desfeita (`rollback`).

`cmax`

O identificador do comando dentro da transação de exclusão, ou zero.

`ctid`

A posição física da versão da linha dentro da tabela. Deve ser observado que, embora seja possível usar `ctid` para localizar a versão da linha muito rapidamente, o `ctid` da linha muda cada vez que a linha é atualizada ou movida pelo comando `VACUUM FULL`. Portanto, o `ctid` não serve como identificador de linha duradouro. O OID ou, melhor ainda, um número serial definido pelo usuário deve ser utilizado para identificar logicamente a linha.

Os OIDs são quantidades de 32 bits atribuídas a partir de um contador único para todo o agrupamento de bancos de dados. Em um banco de dados grande ou existente há muito tempo, é possível que o contador recomece. Portanto, é má prática assumir que os OIDs sejam únicos, a menos que sejam realizados procedimentos para garantir que este seja o caso. Se for necessário identificar as linhas da tabela, recomenda-se a utilização de um gerador de sequência. Entretanto, os OIDs podem ser utilizados desde que sejam tomadas umas poucas precauções adicionais:

- Deve ser criada uma restrição de unicidade na coluna OID de cada tabela onde o OID será utilizado para identificar as linhas.
- Nunca se deve assumir que os OIDs sejam únicos entre tabelas; deve ser utilizada a combinação de `tableoid` com o OID da linha se for necessário um identificador para todo o banco de dados.
- As tabelas em questão devem ser criadas especificando `WITH OIDS`, para garantir a compatibilidade com as versões futuras do PostgreSQL. Planeje-se que `WITHOUT OIDS` se torne o padrão.

Os identificadores das transações também são quantidades de 32 bits. Em um banco de dados existente há muito tempo é possível que os IDs de transação recomecem. Este problema não é fatal se forem obedecidos os procedimentos apropriados de manutenção; consulte o Capítulo 21 para obter detalhes. Entretanto, não é aconselhado depender da unicidade dos IDs de transação por um longo período de tempo (mais de um bilhão de transações).

Os identificadores de comando também são quantidades de 32 bits, criando um limite de  $2^{32}$  (4 bilhões) de comandos SQL dentro de uma única transação. Na prática este limite não é um problema — observe que o limite diz respeito ao número de comandos SQL, e não ao número de linhas processadas.

## 5.5. Herança

Vamos criar duas tabelas. A tabela `capitais` contém as capitais dos estados, que também são cidades. Por consequência, a tabela `capitais` deve herdar da tabela `cidades`.

```
CREATE TABLE cidades (
    nome          text,
    populacao     float,
```

```

    altitude    int        -- (em pés)
);

CREATE TABLE capitais (
    estado      char(2)
) INHERITS (cidades);

```

Neste caso, as linhas da tabela *capitais* *herdam* todos os atributos (nome, população e altitude) de sua tabela ancestral, *cidades*. As capitais dos estados possuem um atributo adicional chamado *estado*, contendo seu estado. No PostgreSQL uma tabela pode herdar de zero ou mais tabelas, e uma consulta pode referenciar tanto todas as linhas de uma tabela, quanto todas as linhas de uma tabela mais todas as linhas de suas descendentes.

**Nota:** A hierarquia de herança é, na verdade, um grafo acíclico dirigido.<sup>6</sup>

Por exemplo, a consulta abaixo retorna os nomes de todas as cidades, incluindo as capitais dos estados, localizadas a uma altitude superior a 500 pés:

```

SELECT nome, altitude
   FROM cidades
  WHERE altitude > 500;

```

nome	altitude
Las Vegas	2174
Mariposa	1953
Madison	845

Por outro lado, a consulta abaixo retorna todas as cidades situadas a uma altitude superior a 500 pés, que não são capitais de estados:

```

SELECT nome, altitude
   FROM ONLY cidades
  WHERE altitude > 500;

```

nome	altitude
Las Vegas	2174
Mariposa	1953

O termo “ONLY” antes de *cidades* indica que a consulta deve ser executada apenas na tabela *cidades*, sem incluir as tabelas descendentes de *cidades* na hierarquia de herança. Muitos comandos mostrados até agora — *SELECT*, *UPDATE* e *DELETE* — suportam esta notação de “ONLY”.

**Em obsolescência:** Nas versões anteriores do PostgreSQL, o comportamento padrão era não incluir as tabelas descendentes nos comandos. Descobriu-se que isso ocasionava muitos erros, e que também violava o padrão SQL:1999. Na sintaxe antiga, para incluir as tabelas descendentes era necessário anexar um *\** ao nome da tabela. Por exemplo:

```
SELECT * FROM cidades*;
```

Ainda é possível especificar explicitamente a varredura das tabelas descendentes anexando o *\**, assim como especificar explicitamente para não varrer as tabelas descendentes escrevendo “ONLY”. A partir da versão 7.1 o comportamento padrão para nomes de tabelas sem adornos passou a ser varrer as tabelas descendentes também, enquanto antes desta versão o comportamento padrão era não varrer as tabelas descendentes. Para habilitar o comportamento padrão antigo, deve ser definida a opção de configuração *SQL\_Inheritance* como desabilitada como, por exemplo,

```
SET SQL_Inheritance TO OFF;
```

ou definir o parâmetro de configuração *sql\_inheritance*.

Em alguns casos pode-se desejar saber de qual tabela uma determinada linha se origina. Em cada tabela existe uma coluna do sistema chamada *tableoid* que pode informar a tabela de origem:



```
SELECT c.tableoid, c.nome, c.altitude
FROM cidades c
WHERE c.altitude > 500;
```

tableoid	nome	altitude
139793	Las Vegas	2174
139793	Mariposa	1953
139798	Madison	845

Se for tentada a reprodução deste exemplo, os valores numéricos dos OIDs provavelmente serão diferentes. Fazendo uma junção com a tabela “pg\_class” é possível mostrar o nome da tabela:

```
SELECT p.relname, c.nome, c.altitude
FROM cidades c, pg_class p
WHERE c.altitude > 500 AND c.tableoid = p.oid;
```

relname	nome	altitude
cidades	Las Vegas	2174
cidades	Mariposa	1953
capitais	Madison	845

Uma tabela pode herdar de mais de uma tabela ancestral e, neste caso, possuirá a união das colunas definidas nas tabelas ancestrais (além de todas as colunas declaradas especificamente para a tabela filha).

Uma limitação séria da funcionalidade da herança é que os índices (incluindo as restrições de unicidade) e as chaves estrangeiras somente se aplicam a uma única tabela, e não às suas descendentes. Isto é verdade tanto do lado que faz referência quanto do lado que é referenciado na chave estrangeira. Portanto, em termos do exemplo acima:

- Se for declarado `cidades.nome` como sendo `UNIQUE` ou `PRIMARY KEY`, isto não impede que a tabela `capitais` tenha linhas com nomes idênticos aos da tabela `cidades` e, por padrão, estas linhas duplicadas aparecem nas consultas à tabela `cidades`. Na verdade, por padrão, a tabela `capitais` não teria nenhuma restrição de unicidade e, portanto, poderia conter várias linhas com nomes idênticos. Poderia ser adicionada uma restrição de unicidade à tabela `capitais`, mas isto não impediria um nome idêntico na tabela `cidades`.
- De forma análoga, se for especificado `cidades.nome REFERENCES` alguma outra tabela, esta restrição não se propaga automaticamente para a tabela `capitais`. Neste caso, o problema poderia ser contornado adicionando manualmente a restrição `REFERENCES` para a tabela `capitais`.
- Especificar para uma coluna de outra tabela `REFERENCES cidades(nome)` permite à outra tabela conter os nomes das `cidades`, mas não os nomes das `capitais`. Não existe uma maneira boa para contornar este problema.

Estas deficiências deverão, provavelmente, serem corrigidas em alguma versão futura, mas enquanto isso deve haver um cuidado considerável ao decidir se a herança é útil para resolver o problema em questão.

## 5.6. Modificação de tabelas

Quando percebemos, após a tabela ser criada, que foi cometido um erro ou que os requisitos do aplicativo mudaram, é possível remover a tabela e criá-la novamente. Porém, esta opção não é conveniente quando existem dados na tabela, ou se a tabela é referenciada por outros objetos do banco de dados (por exemplo, uma restrição de chave estrangeira); por isso, o PostgreSQL disponibiliza um conjunto de comandos para realizar modificações em tabelas existentes. Deve ser observado que esta operação é conceitualmente distinta da alteração dos dados contidos na tabela, aqui o interesse está em mudar a definição, ou estrutura, da tabela.

É possível:

- Adicionar coluna;
- Remover coluna;
- Adicionar restrição;
- Remover restrição;
- Mudar valor padrão;
- Mudar tipo de dado de coluna;

- Mudar nome de coluna;
- Mudar nome de tabela.

Todas estas atividades são realizadas utilizando o comando *ALTER TABLE*.

### 5.6.1. Adicionar coluna

Para adicionar uma coluna, utiliza-se:

```
ALTER TABLE produtos ADD COLUMN descricao text;
```

Inicialmente a nova coluna é preenchida com o valor padrão especificado, ou nulo se a cláusula *DEFAULT* não for especificada.

Também podem ser definidas, ao mesmo tempo, restrições para a coluna utilizando a sintaxe habitual:

```
ALTER TABLE produtos ADD COLUMN descricao text CHECK (descricao <> '');
```

Na verdade, todas as opções que podem ser aplicadas à descrição da coluna no comando *CREATE TABLE* podem ser utilizadas aqui. Entretanto, tenha em mente que o valor padrão deve satisfazer as restrições especificadas, ou o *ADD* não será bem-sucedido. Como alternativa, as restrições podem ser adicionadas posteriormente (veja abaixo), após a nova coluna ter sido preenchida com dados adequados.

### 5.6.2. Remover coluna

Para remover uma coluna, utiliza-se:

```
ALTER TABLE produtos DROP COLUMN descricao;
```

Os dados presentes na coluna desaparecem. As restrições de tabela que envolvem a coluna também são removidas. Entretanto, se a coluna for referenciada por uma restrição de chave estrangeira de outra tabela, o PostgreSQL não irá remover esta restrição em silêncio. Pode ser autorizada a remoção de tudo que depende da coluna adicionando *CASCADE*:

```
ALTER TABLE produtos DROP COLUMN descricao CASCADE;
```

Consulte a Seção 5.10 para obter uma descrição geral do mecanismo por trás desta operação.

### 5.6.3. Adicionar restrição

É utilizada a sintaxe de restrição de tabela para adicionar uma restrição. Por exemplo:

```
ALTER TABLE produtos ADD CHECK (nome <> '');
ALTER TABLE produtos ADD CONSTRAINT unq_cod_prod UNIQUE (cod_prod);
ALTER TABLE produtos ADD FOREIGN KEY (fk_grupo_produtos) REFERENCES grupo_produtos;
```

Para adicionar a restrição de não nulo, que não pode ser escrita na forma de restrição de tabela, deve ser utilizada a sintaxe:

```
ALTER TABLE produtos ALTER COLUMN cod_prod SET NOT NULL;
```

A restrição será verificada imediatamente, portanto os dados da tabela devem satisfazer a restrição para esta poder ser adicionada.

### 5.6.4. Remover restrição

Para remover uma restrição é necessário conhecer seu nome. Se foi atribuído um nome à restrição é fácil, caso contrário o sistema atribui à restrição um nome gerado que precisa ser descoberto. O comando `\d nome_da_tabela` do *psql* pode ser útil nesta situação; outras interfaces também podem oferecer uma forma de inspecionar os detalhes das tabelas. O comando utilizado para remover restrição é:

```
ALTER TABLE produtos DROP CONSTRAINT nome_da_restricao;
```

(Caso esteja lidando com um nome de restrição gerado, como \$2, não se esqueça de colocar entre aspas para torná-lo um identificador válido).

Da mesma forma que para remover uma coluna, é necessário adicionar `CASCADE` se for desejado remover uma restrição que outro objeto dependa. Um exemplo é a restrição de chave estrangeira, que depende da restrição de unicidade ou de chave primária nas colunas referenciadas.

Esta sintaxe serve igualmente para todos os tipos de restrição, exceto não-nulo. Para remover uma restrição de não-nulo, utiliza-se:

```
ALTER TABLE produtos ALTER COLUMN cod_prod DROP NOT NULL;
```

(Lembre-se que as restrições de não-nulo não possuem nome)

### 5.6.5. Mudar valor padrão da coluna

Para definir um novo valor padrão para a coluna, utiliza-se:

```
ALTER TABLE produtos ALTER COLUMN preco SET DEFAULT 7.77;
```

Deve ser observado que este comando não afeta nenhuma coluna existente na tabela, apenas muda o valor padrão para os próximos comandos `INSERT`.

Para remover o valor padrão para a coluna, utiliza-se:

```
ALTER TABLE produtos ALTER COLUMN preco DROP DEFAULT;
```

Efetivamente é o mesmo que definir o valor nulo como sendo o valor padrão. Como consequência, não é errado remover um valor padrão que não tenha sido definido, porque implicitamente o valor nulo é o valor padrão.

### 5.6.6. Mudar o tipo de dado da coluna

Para converter a coluna em um tipo de dado diferente, utiliza-se:

```
ALTER TABLE produtos ALTER COLUMN preco TYPE numeric(10,2);
```

Este comando somente será bem-sucedido se todas as entradas existentes na coluna puderem ser convertidas para o novo tipo através de uma conversão implícita. Se for necessária uma conversão mais complexa, pode ser adicionada a cláusula `USING` especificando como calcular os novos valores a partir dos antigos.

O PostgreSQL tenta converter o valor padrão da coluna (se houver) para o novo tipo, assim bem como todas as restrições que envolvem a coluna. Mas estas conversões podem falhar, ou podem produzir resultados surpreendentes. Geralmente é melhor remover todas as restrições da coluna antes de alterar o seu tipo, e depois adicionar novamente estas restrições modificadas de forma apropriada.

### 5.6.7. Mudar nome de coluna

Para mudar o nome de uma coluna, utiliza-se:

```
ALTER TABLE produtos RENAME COLUMN cod_prod TO cod_produto;
```

### 5.6.8. Mudar nome de tabela

Para mudar o nome de uma tabela, utiliza-se:

```
ALTER TABLE produtos RENAME TO equipamentos;
```

## 5.7. Privilégios

Quem cria o objeto no banco de dados se torna o seu dono. Por padrão, apenas o dono do objeto pode fazer qualquer coisa com o objeto. Para permitir outros usuários utilizarem o objeto, devem ser concedidos *privilégios* (entretanto, os usuários que possuem o atributo de superusuário sempre podem acessar qualquer objeto).

Existem vários privilégios diferentes: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `RULE`, `REFERENCES`, `TRIGGER`, `CREATE`, `TEMPORARY`, `EXECUTE` e `USAGE`. Os privilégios aplicáveis a um determinado tipo de objeto variam de acordo com o tipo do objeto (tabela, função, etc.). Para obter informações completas sobre os diferentes tipos de privilégio suportados pelo

PostgreSQL, deve ser consultada a página de referência do comando *GRANT*. As próximas seções e capítulos também mostram como os privilégios são utilizados.

O direito de modificar e destruir um objeto são sempre privilégios exclusivos do seu criador.

**Nota:** Para mudar o dono de uma tabela, índice, seqüência ou visão deve ser utilizado o comando *ALTER TABLE*. Existem comandos *ALTER* correspondentes para outros tipos de objeto.

Para conceder privilégios utiliza-se o comando *GRANT*. Por exemplo, se *joel* for um usuário existente, e *contas* for uma tabela existente, o privilégio de poder atualizar esta tabela pode ser concedido por meio do comando:

```
GRANT UPDATE ON contas TO joel;
```

Para conceder o privilégio para um grupo é utilizada a sintaxe:

```
GRANT SELECT ON contas TO GROUP contabilidade;
```

O nome especial de “usuário” *PUBLIC* pode ser utilizado para conceder privilégios para todos os usuários do sistema. Escrever *ALL* no lugar do nome específico do privilégio concede todos os privilégios relevantes para o tipo do objeto.

Para revogar um privilégio utiliza-se o comando *REVOKE*:

```
REVOKE ALL ON contas FROM PUBLIC;
```

Os privilégios especiais do dono da tabela (ou seja, os direitos de *DROP*, *GRANT*, *REVOKE*, etc.) são sempre inerentes à condição de ser o dono, não podendo ser concedidos ou revogados. Porém, o dono do objeto pode decidir revogar seus próprios privilégios comuns como, por exemplo, tornar a tabela somente para leitura para o próprio, assim como para os outros.

Normalmente, somente o dono do objeto (ou um superusuário) pode conceder ou revogar privilégios para um objeto. Entretanto, é possível conceder um privilégio “com a opção de concessão”, o que dá a quem recebe o direito de conceder o privilégio para outros. Se mais tarde esta opção de concessão for revogada, então todos aqueles que receberam o privilégio a partir desta pessoa (diretamente ou através de uma cadeia de concessões), perdem este privilégio. Para obter mais detalhes consulte as páginas de referência dos comandos *GRANT* e *REVOKE*.

## 5.8. Esquemas

Um agrupamento de bancos de dados do PostgreSQL contém um ou mais bancos de dados com nome. Os usuários e os grupos de usuários são compartilhados por todo o agrupamento, mas nenhum outro dado é compartilhado entre os bancos de dados. Todas as conexões dos clientes com o servidor podem acessar somente os dados de um único banco de dados, aquele que foi especificado no pedido de conexão.

**Nota:** Os usuários de um agrupamento de bancos de dados não possuem, necessariamente, o privilégio de acessar todos os bancos de dados do agrupamento. O compartilhamento de nomes de usuários significa que não pode haver, em dois bancos de dados do mesmo agrupamento, mais de um usuário com o mesmo nome como, por exemplo, *joel*; mas o sistema pode ser configurado para permitir que o usuário *joel* acesse apenas determinados bancos de dados.

Um banco de dados contém um ou mais *esquemas* com nome, os quais por sua vez contêm tabelas. Os esquemas também contêm outros tipos de objetos com nome, incluindo tipos de dado, funções e operadores. O mesmo nome de objeto pode ser utilizado em esquemas diferentes sem conflito; por exemplo, tanto o *esquema\_1* quanto o *meu\_esquema* podem conter uma tabela chamada *minha\_tabela*. Diferentemente dos bancos de dados, os esquemas não são separados rigidamente: um usuário pode acessar objetos de vários esquemas no banco de dados em que está conectado, caso possua os privilégios necessários para fazê-lo.

Existem diversas razões pelas quais pode-se desejar utilizar esquemas:

- Para permitir vários usuários utilizarem o mesmo banco de dados sem que um interfira com o outro.
- Para organizar objetos do banco de dados em grupos lógicos tornando-os mais gerenciáveis.
- Os aplicativos desenvolvidos por terceiros podem ser colocados em esquemas separados, para não haver colisão com nomes de outros objetos.

Os esquemas são análogos a diretórios no nível do sistema operacional, exceto que os esquemas não podem ser aninhados.

### 5.8.1. Criação de esquema

Para criar um esquema utiliza-se o comando `CREATE SCHEMA`. O nome do esquema é escolhido livremente pelo usuário. Por exemplo:

```
CREATE SCHEMA meu_esquema;
```

Para criar ou acessar objetos em um esquema deve ser escrito um *nome qualificado*, formado pelo nome do esquema e pelo nome da tabela separados por um ponto:

```
nome_do_esquema.nome_da_tabela
```

Esta forma funciona em qualquer local onde é esperado o nome de uma tabela, inclusive nos comandos de modificação de tabela e nos comandos de acesso a dado mostrados nos próximos capítulos (Para abreviar falaremos apenas das tabelas, mas a mesma idéia se aplica a outros tipos de objetos com nome, tais como tipos e funções).

Na verdade, também pode ser utilizada a sintaxe mais geral

```
nome_do_banco_de_dados.nome_do_esquema.nome_da_tabela
```

mas atualmente é apenas uma conformidade *pró-forma* com o padrão SQL; se for escrito o nome do banco de dados, este deverá ter o mesmo nome do banco de dados que se está conectado.

Portanto, para criar uma tabela no novo esquema utiliza-se:

```
CREATE TABLE meu_esquema.minha_tabela (
    ...
);
```

Para remover um esquema vazio (todos os seus objetos já foram removidos), utiliza-se:

```
DROP SCHEMA meu_esquema;
```

Para remover um esquema junto com todos os objetos que este contém, utiliza-se:

```
DROP SCHEMA meu_esquema CASCADE;
```

Consulte a Seção 5.10 para ver a descrição do mecanismo geral por trás desta operação.

Muitas vezes deseja-se criar um esquema cujo dono é outro usuário (porque este é um dos modos utilizados para restringir as atividades dos usuários a espaços de nomes bem definidos). A sintaxe para esta operação é:

```
CREATE SCHEMA nome_do_esquema AUTHORIZATION nome_do_usuario;
```

Inclusive, o nome do esquema pode ser omitido e, neste caso, o nome do esquema será idêntico ao nome do usuário. Consulte a Seção 5.8.6 para ver como pode ser útil.

Os nomes de esquemas começando por `pg_` são reservados para uso pelo sistema, não devendo ser criados pelos usuários.

### 5.8.2. O esquema público

Nas seções anteriores foram criadas tabelas sem que fosse especificado nenhum nome de esquema. Por padrão, estas tabelas (e outros objetos) são colocadas automaticamente no esquema chamado “public”. Todo banco de dados novo possui este esquema. Portanto, as duas formas abaixo são equivalentes:

```
CREATE TABLE produtos ( ... );
```

e

```
CREATE TABLE public.produtos ( ... );
```

### 5.8.3. O caminho de procura do esquema

Os nomes qualificados são desagradáveis de escrever, sendo geralmente melhor não ligar o aplicativo a um esquema específico. Por isso, geralmente as tabelas são referenciadas por meio de *nomes não qualificados*, formados apenas pelo nome da tabela. O sistema determina qual tabela está sendo referenciada seguindo o *caminho de procura*, o qual é uma lista de esquemas para procura. A primeira tabela correspondente encontrada no caminho de procura é assumida como sendo a desejada. Não havendo nenhuma correspondência no caminho de procura é relatado um erro, mesmo que uma tabela correspondendo ao nome exista em outro esquema no banco de dados.

O primeiro nome de esquema no caminho de procura é chamado de esquema corrente. Além de ser o primeiro esquema a ser procurado, também é o esquema onde as novas tabelas são criadas quando o comando `CREATE TABLE` não especifica o nome do esquema.

Para mostrar o caminho de procura corrente, utiliza-se:

```
SHOW search_path;
```

Na configuração padrão este comando retorna:

```
search_path
-----
$user,public
```

O primeiro elemento especifica que deve ser procurado o esquema com o mesmo nome do usuário corrente. Se este esquema não existir, esta entrada é ignorada. O segundo elemento se refere ao esquema público visto anteriormente.

O primeiro esquema no caminho de procura, que exista, é o local padrão para a criação dos novos objetos. Esta é a razão pela qual, por padrão, os objetos são criados no esquema público. Quando os objetos são referenciados em qualquer outro contexto sem qualificação pelo esquema (comandos de modificação de tabelas, modificação de dados ou consultas) o caminho de procura é percorrido até que o objeto correspondente seja encontrado. Portanto, na configuração padrão, qualquer acesso não qualificado somente pode fazer referência ao esquema público.

Para incluir um novo esquema no caminho, utiliza-se:

```
SET search_path TO meu_esquema,public;
```

(O esquema `$user` foi omitido, porque não há necessidade imediata dele). Dessa forma, a tabela pode ser acessada sem ser qualificada pelo esquema:

```
DROP TABLE minha_tabela;
```

Também, como `meu_esquema` é o primeiro elemento do caminho, os novos objetos serão criados neste esquema por padrão.

Também poderia ter sido escrito

```
SET search_path TO meu_esquema;
```

para retirar o acesso ao esquema público sem uma qualificação explícita. Não existe nada especial com relação ao esquema público, a não ser que existe por padrão. Também pode ser excluído.

Consulte também a Seção 9.19 para conhecer outras formas de manipular o caminho de procura de esquema.

O caminho de procura funciona para nomes de tipos de dado, nomes de funções e nomes de operadores, da mesma maneira que funciona para nomes de tabelas. Os nomes dos tipos de dado e das funções podem ser qualificados exatamente da mesma maneira que os nomes das tabelas. Se for necessário escrever um nome de operador qualificado em uma expressão, existe uma maneira especial de fazê-lo, deve ser escrito:

```
OPERATOR(nome_do_esquema.nome_do_operador)
```

Isto é necessário para evitar uma ambigüidade de sintaxe. Por exemplo:

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

Na prática geralmente confia-se no caminho de procura para os operadores, não sendo necessário escrever algo tão horrível assim.

#### 5.8.4. Esquemas e privilégios

Por padrão, os usuários não podem acessar objetos em esquemas que não são seus. Para poderem acessar, o dono do esquema precisa conceder o privilégio `USAGE` para o esquema. Para permitir os usuários utilizarem os objetos do esquema é necessário conceder privilégios adicionais, conforme seja apropriado para cada objeto.

Pode ser permitido, também, que um usuário crie objetos no esquema de outro usuário. Para permitir que isto seja feito, deve ser concedido o privilégio `CREATE` para o esquema. Deve ser observado que, por padrão, todos os usuários possuem o privilégio `CREATE` e `USAGE` para o esquema `public`. Isto permite a todos os usuários que podem se conectar ao banco de dados criar objetos no esquema `public`. Se isto não for desejado, este privilégio pode ser revogado:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

O primeiro “public” acima é o nome do esquema, enquanto o segundo “public” significa “todos os usuários”. Na primeira ocorrência é um identificador, enquanto na segunda ocorrência é uma palavra chave; por isso, na primeira vez está escrito em minúsculas enquanto na segunda vez está em maiúsculas; lembre-se da convenção da Seção 4.1.1.

#### 5.8.5. O esquema do catálogo do sistema

Além do esquema `public` e dos esquemas criados pelos usuários, cada banco de dados contém o esquema `pg_catalog`, contendo as tabelas do sistema e todos os tipos de dado, funções e operadores nativos. O `pg_catalog` é sempre parte efetiva do caminho de procura. Se não for colocado explicitamente no caminho de procura, então é implicitamente procurado *antes* dos esquemas do caminho de procura. Isto garante que os nomes nativos sempre podem ser encontrados. Entretanto, é possível colocar explicitamente o `pg_catalog` no final do caminho de procura, se for desejado que os nomes definidos pelo usuário substituam os nomes nativos.

Nas versões do PostgreSQL anteriores a 7.3, os nomes de tabela começando por `pg_` eram reservados. Isto não é mais verdade: podem ser criadas tabelas com este nome, se for desejado, em qualquer esquema que não seja o do sistema. Entretanto, é melhor continuar evitando estes nomes, para garantir que não haverá conflito caso alguma versão futura defina uma tabela do sistema com o mesmo nome da tabela criada (com o caminho de procura padrão, uma referência não qualificada à tabela criada será resolvida com a tabela do sistema). As tabelas do sistema vão continuar seguindo a convenção de possuir nomes começando por `pg_`, não conflitando com os nomes não qualificados das tabelas dos usuários, desde que os usuários evitem utilizar o prefixo `pg_`.

#### 5.8.6. Modelos de utilização

Os esquemas podem ser utilizados para organizar os dados de várias maneiras. Existem uns poucos modelos de utilização recomendados, facilmente suportados pela configuração padrão:

- Se não for criado nenhum esquema, então todos os usuários acessam o esquema público implicitamente, simulando a situação onde os esquemas não estão disponíveis. Esta configuração é recomendada, principalmente, quando existe no banco de dados apenas um usuário, ou alguns poucos usuários colaborativos. Esta configuração também permite uma transição suave de uma situação sem esquemas.
- Pode ser criado um esquema para cada usuário com o mesmo nome do usuário. Lembre-se que o caminho de procura padrão começa por `$user`, que é resolvido como o nome do usuário. Portanto, se cada usuário possuir um esquema separado, vão acessar seus próprios esquemas por padrão.

Se esta configuração for utilizada, também pode ser revogado o acesso ao esquema público (ou mesmo removê-lo), deixando os usuários totalmente restritos aos seus próprios esquemas.

- Para instalar aplicações compartilhadas (tabelas utilizadas por todos, funções adicionais fornecidas por terceiros, etc.), estas devem ser colocadas em esquemas separados. Devem ser concedidos, também, os privilégios necessários para permitir o acesso pelos outros usuários. Os usuários poderão, então, fazer referência a estes objetos adicionais qualificando seus nomes com o nome do esquema, ou poderão adicionar esquemas ao caminho de procura, conforme julgarem melhor.

### 5.8.7. Portabilidade

No padrão SQL, não existe a noção de objetos no mesmo esquema pertencendo a usuários diferentes. Além disso, algumas implementações não permitem criar esquemas com nome diferente do nome de seu dono. Na verdade, os conceitos de esquema e de usuário são praticamente equivalentes em sistemas de banco de dados que implementam somente o suporte básico a esquemas especificado no padrão. Portanto, muitos usuários consideram os nomes qualificados na verdade formados por *nome\_do\_usuario.nome\_da\_tabela*. Esta é a forma como o PostgreSQL se comportará efetivamente, se for criado um esquema por usuário para todos os usuários.

Além disso, não existe o conceito do esquema `public` no padrão SQL. Para máxima conformidade com o padrão, o esquema `public` não deve ser utilizado (talvez deva até ser removido).

Obviamente, alguns sistemas de banco de dados SQL podem não implementar esquemas de nenhuma maneira, ou oferecer suporte a espaços de nomes permitindo apenas acesso entre bancos de dados (possivelmente limitado). Se for necessário trabalhar com estes sistemas, o máximo de portabilidade é obtido não utilizando nada relacionado a esquemas.<sup>7 8 9</sup>

### 5.8.8. Exemplos

**Nota:** Seção escrita pelo tradutor, não fazendo parte do manual original.

#### Exemplo 5-5. Informações sobre esquema

Este exemplo mostra a utilização do comando `\dn` do `psql` e das funções `current_schema()` e `current_schemas()` para obter informações sobre esquema.

```
=> \dn
      Lista de esquemas
      Nome           | Dono
      -----+-----
information_schema | postgres
pg_catalog         | postgres
pg_toast           | postgres
public             | postgres
(4 linhas)
```

```
=> \df current_schema*
      Lista de funções
      Esquema | Nome           | Tipo de dado do resultado | Tipo de dado dos argumentos
      -----+-----+-----+-----
pg_catalog | current_schema | name                      |
pg_catalog | current_schemas | name[]                    | boolean
(2 linhas)
```

```
=> SELECT current_schema();

current_schema
-----
public
(1 linha)
```

```
=> SELECT current_schemas(true);

current_schemas
-----
{pg_catalog,public}
(1 linha)
```

```
=> SELECT current_schemas(false);

current_schemas
-----
{public}
(1 linha)
```



## 5.9. Outros objetos de banco de dados

As tabelas são os objetos centrais da estrutura de um banco de dados relacional, porque armazenam os dados, mas não são os únicos objetos que existem no banco de dados. Podem ser criados vários objetos de outros tipos, para tornar o uso e o gerenciamento dos dados mais eficiente, ou mais conveniente. Estes outros objetos não são mostrados neste capítulo, mas são listados abaixo para que se tome conhecimento do que é possível criar.

- Visões
- Funções e operadores
- Tipos de dado e domínios
- Gatilhos e regras de reescrita

Informações detalhadas sobre estes tópicos são mostradas na Parte V.

## 5.10. Acompanhando as dependências

Ao se criar uma estrutura de banco de dados complexa, envolvendo muitas tabelas com restrições de chave estrangeira, visões, gatilhos, funções, etc., cria-se, implicitamente, uma rede de dependências entre os objetos. Por exemplo, uma tabela com uma restrição de chave estrangeira depende da tabela referenciada.

Para garantir a integridade de toda a estrutura do banco de dados, o PostgreSQL não permite remover um objeto quando há objetos que dependem do mesmo. Por exemplo, tentar remover a tabela `produtos`, conforme declarada na Seção 5.3.5 onde a tabela `pedidos` depende dela, produz uma mensagem de erro como esta:

```
DROP TABLE produtos;
```

```
NOTA: a restrição pedidos_cod_prod_fkey na tabela pedidos depende da tabela produtos
ERRO: não foi possível remover a tabela produtos porque outros objetos dependem da mesma
DICA: Use DROP ... CASCADE para remover os objetos dependentes também.
```

A mensagem de erro mostra uma dica útil: Se não tem importância remover todos os objetos dependentes, então pode ser executado

```
DROP TABLE produtos CASCADE;
```

e todos os objetos dependentes serão removidos. Neste caso não será removida a tabela `pedidos`, será removida apenas a restrição de chave estrangeira (caso se deseje verificar o que `DROP ... CASCADE` fará, deve ser executado o comando `DROP` sem o `CASCADE`, e lidas as NOTAS, ou NOTICE em inglês).

Todos os comandos de remoção do PostgreSQL permitem especificar `CASCADE`. Obviamente, a natureza das dependências possíveis varia conforme o tipo do objeto. Pode ser escrito `RESTRICT` em vez de `CASCADE`, para obter o comportamento padrão que é impedir a remoção do objeto quando existem objetos que dependem do mesmo.

**Nota:** De acordo com o padrão SQL é obrigatório especificar `RESTRICT` ou `CASCADE`. Nenhum banco de dados obriga seguir esta regra, mas tornar `RESTRICT` ou `CASCADE` o comportamento padrão varia entre sistemas.

**Nota:** As dependências de restrição de chave estrangeira e as dependências de coluna serial, das versões do PostgreSQL anteriores a 7.3, *não* são mantidas ou criadas durante o processo de atualização de versão. Todos os outros tipos de dependência são criados de forma apropriada durante a atualização de uma versão anterior a 7.3.

## Notas

1. *semântica* — do Gr. *semantiké*, da significação — estudo da linguagem humana do ponto de vista do significado das palavras e dos enunciados. PRIBERAM - Língua Portuguesa On-Line (<http://www.priberam.pt/dlpo/dlpo.aspx>). (N. do T.)
2. *truth-value* — Na lógica, o valor verdade, ou valor-verdade, é um valor indicando até que ponto uma declaração é verdadeira; Na lógica clássica, os únicos valores verdade possíveis são verdade e falso. Entretanto, são possíveis outros valores em outras lógicas. A lógica intuicionista simples possui os valores verdade: verdade, falso e desconhecido. A lógica fuzzy, e outras formas de lógica multi-valoradas, também possuem mais valores verdade do que simplesmente

verdade e falso; Algebricamente, o conjunto {verdade, falso} forma a lógica booleana simples. Dictionary.LaborLawTalk.com ([http://encyclopedia.laborlawtalk.com/Truth\\_value](http://encyclopedia.laborlawtalk.com/Truth_value)) (N. do T.)

3. Oracle 9i — Para satisfazer a restrição de unicidade, não podem haver duas linhas na tabela com o mesmo valor para a chave única. Entretanto, a chave única formada por uma única coluna pode conter nulos. Para satisfazer uma chave única composta, não podem haver duas linhas na tabela ou na visão com a mesma combinação de valores nas colunas chave. Qualquer linha contendo nulo em todas as colunas chave satisfaz, automaticamente, a restrição. Entretanto, duas linhas contendo nulo em uma ou mais colunas chave, e a mesma combinação de valores para as outras colunas chave, violam a restrição. Oracle9i SQL Reference (<http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/clauses3a.htm>)
4. SQL Server 2000 — As restrições de unicidade podem ser utilizadas para garantir que não serão entrados valores duplicados em colunas específicas que não participam da chave primária. Embora tanto a restrição UNIQUE quanto a restrição PRIMARY KEY obriguem a unicidade, deve ser utilizado UNIQUE em vez de PRIMARY KEY quando se deseja garantir a unicidade de: uma coluna, ou combinação de colunas, que não seja a chave primária (podem ser definidas várias restrições UNIQUE em uma tabela, mas somente uma restrição PRIMARY KEY); uma coluna que aceite o valor nulo (as restrições UNIQUE podem ser definidas em colunas que permitem o valor nulo, enquanto as restrições PRIMARY KEY somente podem ser definidas em colunas que não aceitam o valor nulo). — A restrição UNIQUE também pode ser referenciada pela restrição FOREIGN KEY. — Quando é adicionada uma restrição UNIQUE a uma coluna, ou colunas, existentes em uma tabela, os dados existentes nas colunas são verificados para garantir que todos os valores, exceto os nulos, são únicos. SQL Server Books Online (N. do T.)
5. DB2 8.1 — A restrição de unicidade é a regra que especifica que os valores de uma chave são válidos apenas se forem únicos na tabela. As colunas especificadas em uma restrição de unicidade devem ser definidas como NOT NULL. O gerenciador de banco de dados usa um índice único para obrigar a unicidade da chave durante as alterações nas colunas da restrição de unicidade. A restrição de unicidade que é referenciada por uma chave estrangeira de uma restrição referencial é chamada de chave pai. Deve ser observado que existe uma distinção entre definir uma restrição de unicidade e criar um índice único: embora ambos obriguem a unicidade, o índice único permite colunas com valor nulo e, geralmente, não pode ser utilizado como uma chave pai. Unique Constraints (<https://aurora.vcu.edu/db2help/db2s0/frame3.htm#cnstrnt>) (N. do T.)
6. Grafo: uma coleção de vértices e arestas; Grafo dirigido: um grafo com arestas unidirecionais; Grafo acíclico dirigido: um grafo dirigido que não contém ciclos - FOLDOC - Free On-Line Dictionary of Computing (<http://wombat.doc.ic.ac.uk/foldoc/index.html>) (N. do T.)
7. Oracle 9i — Um esquema é uma coleção de objetos de banco de dados. O esquema pertence ao usuário do banco de dados e possui o mesmo nome do usuário. Os objetos do esquema são estruturas lógicas que se referem diretamente aos dados do banco de dados. Os objetos do esquema incluem estruturas como tabelas, visões e índices. Não existe relacionamento entre espaço de tabela e esquema; objetos do mesmo esquema podem estar em espaços de tabela diferentes, e espaços de tabelas podem conter objetos de esquemas diferentes. Introduction to the Oracle Server ([http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96524/c01\\_02intro.htm](http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96524/c01_02intro.htm))
8. SQL Server 2000 — O nome completo de um objeto é formado por quatro identificadores: o nome do servidor, o nome do banco de dados, o nome do dono e o nome do objeto. Aparecem no seguinte formato: [ [ [ nome\_do\_servidor. ] [ nome\_do\_banco\_de\_dados ] . ] [ nome\_do\_dono ] . ] nome\_do\_objeto. O nome do servidor, do banco de dados e do dono são conhecidos como qualificadores do nome do objeto. SQL Server Books Online (N. do T.)
9. DB2 8.1 — Um esquema é uma coleção de objetos com nome. Os esquemas provêm uma classificação lógica dos objetos no banco de dados. O esquema pode conter tabelas, visões, apelidos, gatilhos, funções, pacotes e outros objetos. O esquema também é um objeto do banco de dados. O nome do esquema é utilizado como a parte de mais alta ordem de um nome de objeto de duas partes. Se o objeto for qualificado com um nome de esquema específico ao ser criado, o objeto é atribuído a este esquema. Se não for especificado nenhum nome de esquema ao criar um objeto, é utilizado o nome de esquema padrão. Os esquemas também possuem privilégios, permitindo ao dono do esquema controlar quais usuários possuem o privilégio de criar, alterar e remover objetos no esquema. Schemas (<https://aurora.vcu.edu/db2help/db2s0/frame3.htm#sqlschem>) (N. do T.)

# Capítulo 6. Manipulação de dados

O capítulo anterior mostrou como criar tabelas e outras estruturas para armazenar dados. Agora está na hora de preencher as tabelas com dados. Este capítulo mostra como inserir, atualizar e excluir dados em tabelas. Também são apresentadas maneiras de efetuar mudanças automáticas nos dados quando ocorrem certos eventos: gatilhos (*triggers*) e regras de reescrita (*rewrite rules*). Para completar, o próximo capítulo explica como fazer consultas para extrair dados do banco de dados.

## 6.1. Inserção de dados

A tabela recém-criada não contém dados. A primeira ação a ser realizada para o banco de dados ter utilidade é inserir dados. Conceitualmente, os dados são inseridos uma linha de cada vez. É claro que é possível inserir mais de uma linha, mas não existe maneira de inserir menos de uma linha por vez. Mesmo que se conheça apenas o valor de algumas colunas, deve ser criada uma linha completa.

Para criar uma linha é utilizado o comando `INSERT`. Este comando requer o nome da tabela, e um valor para cada coluna da tabela. Por exemplo, considere a tabela `produtos` do Capítulo 5:

```
CREATE TABLE produtos (  
    cod_prod    integer,  
    nome        text,  
    preco       numeric  
);
```

Um exemplo de comando para inserir uma linha é:

```
INSERT INTO produtos VALUES (1, 'Queijo', 9.99);
```

Os valores dos dados são colocados na mesma ordem que as colunas se encontram na tabela, separados por vírgula. Geralmente os valores dos dados são literais (constantes), mas também são permitidas expressões escalares.

A sintaxe mostrada acima tem como desvantagem ser necessário conhecer a ordem das colunas da tabela. Para evitar isto, as colunas podem ser relacionadas explicitamente. Por exemplo, os dois comandos mostrados abaixo possuem o mesmo efeito do comando mostrado acima:

```
INSERT INTO produtos (cod_prod, nome, preco) VALUES (1, 'Queijo', 9.99);  
INSERT INTO produtos (nome, preco, cod_prod) VALUES ('Queijo', 9.99, 1);
```

Muitos usuários consideram boa prática escrever sempre os nomes das colunas.

Se não forem conhecidos os valores de todas as colunas, as colunas com valor desconhecido podem ser omitidas. Neste caso, estas colunas são preenchidas com seu respectivo valor padrão. Por exemplo:

```
INSERT INTO produtos (cod_prod, nome) VALUES (1, 'Queijo');  
INSERT INTO produtos VALUES (1, 'Queijo');
```

A segunda forma é uma extensão do PostgreSQL, que preenche as colunas a partir da esquerda com quantos valores forem fornecidos, e as demais com o valor padrão.

Para ficar mais claro, pode ser requisitado explicitamente o valor padrão da coluna individualmente, ou para toda a linha:

```
INSERT INTO produtos (cod_prod, nome, preco) VALUES (1, 'Queijo', DEFAULT);  
INSERT INTO produtos DEFAULT VALUES;
```

**Dica:** Para realizar “cargas volumosas”, ou seja, inserir muitos dados, consulte o comando *COPY*. Este comando não é tão flexível quanto o comando `INSERT`, mas é mais eficiente.

## 6.2. Atualização de dados

A modificação dos dados armazenados no banco de dados é referida como atualização. Pode ser atualizada uma linha, todas as linhas, ou um subconjunto das linhas da tabela. Uma coluna pode ser atualizada separadamente; as outras colunas não são afetadas.

Para realizar uma atualização são necessárias três informações:

1. O nome da tabela e da coluna a ser atualizada;
2. O novo valor para a coluna;
3. Quais linhas serão atualizadas.

Lembre-se que foi dito no Capítulo 5 que o SQL, de uma maneira geral, não fornece um identificador único para as linhas. Portanto, não é necessariamente possível especificar diretamente a linha a ser atualizada. Em vez disso, devem ser especificadas as condições que a linha deve atender para ser atualizada. Somente havendo uma chave primária na tabela (não importando se foi declarada ou não), é possível endereçar uma linha específica com confiança, escolhendo uma condição correspondendo à chave primária. Ferramentas gráficas de acesso a banco de dados dependem da chave primária para poderem atualizar as linhas individualmente.

Por exemplo, o comando mostrado abaixo atualiza todos os produtos com preço igual a 5, mudando estes preços para 10:

```
UPDATE produtos SET preco = 10 WHERE preco = 5;
```

Este comando pode atualizar nenhuma, uma, ou muitas linhas. Não é errado tentar uma atualização que não corresponda a nenhuma linha.

Vejamos este comando em detalhe: Primeiro aparece a palavra chave `UPDATE` seguida pelo nome da tabela. Como usual, o nome da tabela pode ser qualificado pelo esquema, senão é procurado no caminho. Depois aparece a palavra chave `SET`, seguida pelo nome da coluna, por um sinal de igual, e pelo novo valor da coluna. O novo valor da coluna pode ser qualquer expressão escalar, e não apenas uma constante. Por exemplo, se for desejado aumentar o preço de todos os produtos em 10% pode ser utilizado:

```
UPDATE produtos SET preco = preco * 1.10;
```

Como pode ser visto, a expressão para obter o novo valor pode fazer referência ao valor antigo. Também foi deixada de fora a cláusula `WHERE`. Quando esta cláusula é omitida, significa que todas as linhas da tabela serão atualizadas e, quando está presente, somente as linhas que atendem à condição desta cláusula serão atualizadas. Deve ser observado que o sinal de igual na cláusula `SET` é uma atribuição, enquanto o sinal de igual na cláusula `WHERE` é uma comparação, mas isto não cria uma ambigüidade. Obviamente, a condição da cláusula `WHERE` não é necessariamente um teste de igualdade, estão disponíveis vários outros operadores (consulte o Capítulo 9), mas a expressão deve produzir um resultado booleano.

Também pode ser atualizada mais de uma coluna pelo comando `UPDATE`, colocando mais de uma atribuição na cláusula `SET`. Por exemplo:

```
UPDATE minha_tabela SET a = 5, b = 3, c = 1 WHERE a > 0;
```

## 6.3. Exclusão de dados

Até aqui foi mostrado como adicionar dados a tabelas, e como modificar estes dados. Está faltando mostrar como remover os dados que não são mais necessários. Assim como só é possível adicionar dados para toda uma linha, uma linha também só pode ser removida por inteiro da tabela. Na seção anterior foi explicado que o SQL não fornece uma maneira para endereçar diretamente uma determinada linha. Portanto, a remoção das linhas só pode ser feita especificando as condições que as linhas a serem removidas devem atender. Havendo uma chave primária na tabela, então é possível especificar exatamente a linha. Mas também pode ser removido um grupo de linhas atendendo a uma determinada condição, ou podem ser removidas todas as linhas da tabela de uma só vez.

É utilizado o comando `DELETE` para remover linhas; a sintaxe deste comando é muito semelhante a do comando `UPDATE`. Por exemplo, para remover todas as linhas da tabela produtos possuindo preço igual a 10:

```
DELETE FROM produtos WHERE preco = 10;
```

Se for escrito simplesmente

```
DELETE FROM produtos;
```

então todas as linhas da tabela serão excluídas! Dica de programador.

# Capítulo 7. Consultas

Os capítulos anteriores explicaram como criar tabelas, como preenchê-las com dados, e como manipular estes dados. Agora, finalmente, é mostrado como trazer estes dados para fora do banco de dados.

## 7.1. Visão geral

O processo de trazer, ou o comando para trazer os dados armazenados no banco de dados, é chamado de *consulta*. No SQL, o comando `SELECT` é utilizado para especificar consultas. A sintaxe geral do comando `SELECT` é

```
SELECT lista_de_seleção FROM expressão_de_tabela [especificação_da_ordenação]
```

As próximas seções descrevem em detalhes a lista de seleção, a expressão de tabela, e a especificação da ordenação.

O tipo mais simples de consulta possui a forma:

```
SELECT * FROM tabela1;
```

Supondo existir uma tabela chamada `tabela1`, este comando traz todas as linhas e todas as colunas da `tabela1`. A forma de trazer depende do aplicativo cliente. Por exemplo, o aplicativo `psql` exibe uma tabela ASCII formatada na tela, enquanto as bibliotecas cliente disponibilizam funções para extrair valores individuais do resultado da consulta. A especificação da lista de seleção `*` significa todas as colunas que a expressão de tabela possa fornecer. A lista de seleção também pode selecionar um subconjunto das colunas disponíveis, ou efetuar cálculos utilizando as colunas. Por exemplo, se a `tabela1` possui colunas chamadas `a`, `b` e `c` (e talvez outras), pode ser feita a seguinte consulta:

```
SELECT a, b + c FROM tabela1;
```

(Supondo que `b` e `c` possuem um tipo de dado numérico). Consulte a Seção 7.3 para obter mais detalhes.

`FROM tabela1` é um tipo particularmente simples de expressão de tabela: lê apenas uma única tabela. De uma forma geral, as expressões de tabela podem ser construções complexas contendo tabelas base, junções e subconsultas. Mas a expressão de tabela pode ser totalmente omitida, quando se deseja utilizar o comando `SELECT` como uma calculadora:

```
SELECT 3 * 4;
```

É mais útil quando as expressões da lista de seleção retornam resultados variáveis. Por exemplo, uma função pode ser chamada deste modo:

```
SELECT random();
```

## 7.2. Expressões de tabela

Uma *expressão de tabela* computa uma tabela. A expressão de tabela contém a cláusula `FROM` seguida, opcionalmente, pelas cláusulas `WHERE`, `GROUP BY` e `HAVING`. As expressões de tabela triviais fazem, simplesmente, referência as tão faladas *tabelas em disco*, chamadas de *tabelas base*, mas podem ser utilizadas expressões mais complexas para modificar ou combinar tabelas base de várias maneiras.

As cláusulas opcionais `WHERE`, `GROUP BY` e `HAVING`, da expressão de tabela, especificam um processo de transformações sucessivas realizadas na tabela produzida pela cláusula `FROM`. Todas estas transformações produzem uma tabela virtual que fornece as linhas passadas para a lista de seleção, para então serem computadas as linhas de saída da consulta.

### 7.2.1. A cláusula FROM

A *Cláusula FROM* deriva uma tabela a partir de uma ou mais tabelas especificadas na lista, separada por vírgulas, de referências a tabela.

```
FROM referência_a_tabela [, referência_a_tabela [, ...]]
```

Uma referência a tabela pode ser um nome de tabela (possivelmente qualificado pelo esquema) ou uma tabela derivada, como uma subconsulta, uma junção de tabelas ou, ainda, uma combinação complexa destas. Se for listada mais de uma referência a tabela na cláusula `FROM`, é feita uma junção cruzada (`CROSS JOIN`) (veja abaixo) para formar a tabela virtual intermediária que poderá, então, estar sujeita às transformações das cláusulas `WHERE`, `GROUP BY` e `HAVING`, gerando o resultado final de toda a expressão de tabela.

Quando uma referência a tabela especifica uma tabela ancestral em uma hierarquia de herança de tabelas, a referência a tabela não produz linhas apenas desta tabela, mas inclui as linhas de todas as tabelas descendentes, a não ser que a palavra chave `ONLY` preceda o nome da tabela. Entretanto, esta referência produz apenas as colunas existentes na tabela especificada — são ignoradas todas as colunas adicionadas às tabelas descendentes.

### 7.2.1.1. Junção de tabelas

Uma tabela juntada é uma tabela derivada de outras duas tabelas (reais ou derivadas), de acordo com as regras do tipo particular de junção. Estão disponíveis as junções internas, externas e cruzadas.

## Tipos de junção

### Junção cruzada

```
T1 CROSS JOIN T2
```

Para cada combinação de linhas de *T1* e *T2*, a tabela derivada contém uma linha formada por todas as colunas de *T1* seguidas por todas as colunas de *T2*. Se as tabelas possuírem *N* e *M* linhas, respectivamente, a tabela juntada terá *N \* M* linhas.

`FROM T1 CROSS JOIN T2` equivale a `FROM T1, T2`. Também equivale a `FROM T1 INNER JOIN T2 ON TRUE` (veja abaixo).

### Junções qualificadas

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON expressão_booleana
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( lista de colunas de
junção )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

As palavras `INNER` e `OUTER` são opcionais em todas as formas. `INNER` é o padrão; `LEFT`, `RIGHT` e `FULL` implicam em junção externa.

A *condição de junção* é especificada na cláusula `ON` ou `USING`, ou implicitamente pela palavra `NATURAL`. A condição de junção determina quais linhas das duas tabelas de origem são consideradas “correspondentes”, conforme explicado detalhadamente abaixo.

A cláusula `ON` é o tipo mais geral de condição de junção: recebe uma expressão de valor booleana do mesmo tipo utilizado na cláusula `WHERE`. Um par de linhas de *T1* e *T2* são correspondentes se a expressão da cláusula `ON` for avaliado como verdade para este par de linhas.

`USING` é uma notação abreviada: recebe uma lista de nomes de colunas, separados por vírgula, que as tabelas juntadas devem possuir em comum, e forma a condição de junção especificando a igualdade de cada par destas colunas. Além disso, a saída de `JOIN USING` possui apenas uma coluna para cada par da igualdade de colunas da entrada, seguidas por todas as outras colunas de cada tabela. Portanto, `USING (a, b, c)` equivale a `ON (t1.a = t2.a AND t1.b = t2.b AND t1.c = t2.c)`, mas quando `ON` é utilizado existem duas colunas *a*, *b* e *c* no resultado, enquanto usando `USING` existe apenas uma de cada.

Finalizando, `NATURAL` é uma forma abreviada de `USING`: gera uma lista `USING` formada pelas colunas cujos nomes aparecem nas duas tabelas de entrada. Assim como no `USING`, estas colunas aparecem somente uma vez na tabela de saída.

Os tipos possíveis de junção qualificada são:

```
INNER JOIN
```

Para cada linha *L1* de *T1*, a tabela juntada possui uma linha para cada linha de *T2* que satisfaz a condição de junção com *L1*.

## LEFT OUTER JOIN

Primeiro, é realizada uma junção interna. Depois, para cada linha de T1 que não satisfaz a condição de junção com nenhuma linha de T2, é adicionada uma linha juntada com valores nulos nas colunas de T2. Portanto, a tabela juntada possui, incondicionalmente, no mínimo uma linha para cada linha de T1.

## RIGHT OUTER JOIN

Primeiro, é realizada uma junção interna. Depois, para cada linha de T2 que não satisfaz a condição de junção com nenhuma linha de T1, é adicionada uma linha juntada com valores nulos nas colunas de T1. É o oposto da junção esquerda: a tabela resultante possui, incondicionalmente, uma linha para cada linha de T2.

## FULL OUTER JOIN

Primeiro, é realizada uma junção interna. Depois, para cada linha de T1 que não satisfaz a condição de junção com nenhuma linha de T2, é adicionada uma linha juntada com valores nulos nas colunas de T2. Também, para cada linha de T2 que não satisfaz a condição de junção com nenhuma linha de T1, é adicionada uma linha juntada com valores nulos nas colunas de T1.

As junções de todos os tipos podem ser encadeadas ou aninhadas: tanto *T1* como *T2*, ou ambas, podem ser tabelas juntadas. Podem colocados parênteses em torno das cláusulas `JOIN` para controlar a ordem de junção. Na ausência de parênteses, as cláusulas `JOIN` são aninhadas da esquerda para a direita.

Para reunir tudo isto, vamos supor que temos as tabelas *t1*

```
num | nome
-----+-----
  1 | a
  2 | b
  3 | c
```

e *t2*

```
num | valor
-----+-----
  1 | xxx
  3 | yyy
  5 | zzz
```

e mostrar os resultados para vários tipos de junção:

```
=> SELECT * FROM t1 CROSS JOIN t2;
```

```
num | nome | num | valor
-----+-----+-----+-----
  1 | a    |    1 | xxx
  1 | a    |    3 | yyy
  1 | a    |    5 | zzz
  2 | b    |    1 | xxx
  2 | b    |    3 | yyy
  2 | b    |    5 | zzz
  3 | c    |    1 | xxx
  3 | c    |    3 | yyy
  3 | c    |    5 | zzz
```

(9 linhas)

```
=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
```

```
num | nome | num | valor
-----+-----+-----+-----
  1 | a    |    1 | xxx
  3 | c    |    3 | yyy
```

(2 linhas)

```
=> SELECT * FROM t1 INNER JOIN t2 USING (num);
```



```

num | nome | valor
-----+-----+-----
  1 | a    | xxx
  3 | c    | yyy
(2 linhas)

```

```
=> SELECT * FROM t1 NATURAL INNER JOIN t2;
```

```

num | nome | valor
-----+-----+-----
  1 | a    | xxx
  3 | c    | yyy
(2 linhas)

```

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
```

```

num | nome | num | valor
-----+-----+-----+-----
  1 | a    |    1 | xxx
  2 | b    |    | 
  3 | c    |    3 | yyy
(3 linhas)

```

```
=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
```

```

num | nome | valor
-----+-----+-----
  1 | a    | xxx
  2 | b    | 
  3 | c    | yyy
(3 linhas)

```

```
=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
```

```

num | nome | num | valor
-----+-----+-----+-----
  1 | a    |    1 | xxx
  3 | c    |    3 | yyy
    |      |    5 | zzz
(3 linhas)

```

```
=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
```

```

num | nome | num | valor
-----+-----+-----+-----
  1 | a    |    1 | xxx
  2 | b    |    | 
  3 | c    |    3 | yyy
    |      |    5 | zzz
(4 linhas)

```

A condição de junção especificada em `ON` também pode conter condições não relacionadas diretamente com a junção. Pode ser útil em algumas consultas, mas deve ser usado com cautela. Por exemplo:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.valor = 'xxx';
```

```

num | nome | num | valor
-----+-----+-----+-----
  1 | a    |    1 | xxx
  2 | b    |    | 
  3 | c    |    | 
(3 linhas)

```

A Tabela 7-1 mostra os tipos de junção suportados pelos gerenciadores de banco de dados PostgreSQL, SQL Server, Oracle e DB2.<sup>1</sup>

**Tabela 7-1. Tipos de junção no PostgreSQL, no SQL Server, no Oracle e no DB2**

Tipo de junção	PostgreSQL 8.0.0	SQL Server 2000	Oracle 10g	DB2 8.1
INNER JOIN ON	sim	sim	sim	sim
LEFT OUTER JOIN ON	sim	sim	sim	sim
RIGHT OUTER JOIN ON	sim	sim	sim	sim
FULL OUTER JOIN ON	sim	sim	sim	sim
INNER JOIN USING	sim	não	sim	não
CROSS JOIN	sim	sim	sim	não
NATURAL JOIN	sim	não	sim	não

### 7.2.1.2. Aliases de tabela e de coluna

Pode ser dado um nome temporário às tabelas, e às referências a tabela complexas, para ser usado nas referências à tabela derivada no restante do comando. Isto é chamado de *aliás de tabela*.<sup>2</sup>

Para criar um aliás de tabela, escreve-se

```
FROM referência_a_tabela AS aliás
```

ou

```
FROM referência_a_tabela aliás
```

A palavra chave AS é opcional. O *aliás* pode ser qualquer identificador.

Uma utilização típica de aliás de tabela é para atribuir identificadores curtos a nomes de tabelas longos, para manter a cláusula de junção legível. Por exemplo:

```
SELECT * FROM um_nome_muito_comprido u JOIN outro_nome_muito_comprido o ON u.id = o.num;
```

O aliás se torna o novo nome da referência à tabela na consulta corrente — não é mais possível fazer referência à tabela pelo seu nome original. Portanto,

```
SELECT * FROM minha_tabela AS m WHERE minha_tabela.a > 5;
```

não é uma sintaxe SQL válida. O que acontece de verdade (isto é uma extensão do PostgreSQL ao padrão), é que uma referência a tabela implícita é adicionada à cláusula FROM. Portanto, a consulta é processada como se tivesse sido escrita assim

```
SELECT * FROM minha_tabela AS m, minha_tabela AS minha_tabela WHERE minha_tabela.a > 5;
```

resultando em uma junção cruzada, que geralmente não é o que se deseja.

Os aliases de tabela servem principalmente como uma notação conveniente, mas sua utilização é necessária para fazer a junção de uma tabela consigo mesma. Por exemplo:

```
SELECT * FROM minha_tabela AS a CROSS JOIN minha_tabela AS b ...
```

Além disso, um aliás é requerido se a referência a tabela for uma subconsulta (consulte a Seção 7.2.1.3).

Os parênteses são utilizados para resolver ambigüidades. A declaração abaixo atribui o aliás b ao resultado da junção, diferentemente do exemplo anterior:

```
SELECT * FROM (minha_tabela AS a CROSS JOIN minha_tabela) AS b ...
```

Uma outra forma de aliás de tabela especifica nomes temporários para as colunas da tabela, assim como para a mesma:

```
FROM referência_a_tabela [AS] aliás ( coluna1 [, coluna2 [, ...]] )
```

Se for especificado um número de aliases de coluna menor que o número de colunas da tabela, as demais colunas não terão o nome mudado. Esta sintaxe é especialmente útil em autojunções e subconsultas.

Quando um aliás é aplicado à saída da cláusula JOIN, utilizando qualquer uma destas formas, o aliás esconde o nome original dentro do JOIN. Por exemplo:

```
SELECT a.* FROM minha_tabela AS a JOIN sua_tabela AS b ON ...
```

é um comando SQL válido, mas

```
SELECT a.* FROM (minha_tabela AS a JOIN sua_tabela AS b ON ...) AS c
```

não é válido: o aliás de tabela a não é visível fora do aliás c.

### 7.2.1.3. Subconsultas

Subconsultas especificando uma tabela derivada devem estar entre parênteses, e *devem* ter um nome de aliás de tabela atribuído (consulte a Seção 7.2.1.2). Por exemplo:

```
FROM (SELECT * FROM tabela1) AS nome_aliás
```

Este exemplo equivale a FROM tabela1 AS nome\_aliás. Casos mais interessantes, que não podem ser reduzidos a junções simples, ocorrem quando a subconsulta envolve agrupamento ou agregação.

### 7.2.1.4. Funções de tabela

As funções de tabela são funções que produzem um conjunto de linhas, formadas por um tipo de dado base (tipos escalar), ou por um tipo de dado composto (linhas de tabela). São utilizadas como uma tabela, visão ou subconsulta na cláusula FROM da consulta. As colunas retornadas pelas funções de tabela podem ser incluídas nas cláusulas SELECT, JOIN ou WHERE da mesma maneira que uma coluna de tabela, visão ou de subconsulta.

Se a função de tabela retornar um tipo de dado base, a única coluna do resultado recebe o nome da função. Se a função retornar um tipo composto, as colunas do resultado recebem o mesmo nome dos atributos individuais do tipo.

A função de tabela pode receber um aliás na cláusula FROM, mas também pode ser deixada sem aliás. Se a função for utilizada na cláusula FROM sem aliás, o nome da função é utilizado como o nome da tabela resultante.

Alguns exemplos:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
```

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM getfoo(1) AS t1;
```

```
SELECT * FROM foo
    WHERE foosubid IN (select foosubid from getfoo(foo.fooid) z
                      where z.fooid = foo.fooid);
```

```
CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);
```

```
SELECT * FROM vw_getfoo;
```

Em alguns casos é útil definir funções de tabela que possam retornar conjuntos de colunas diferentes dependendo de como são chamadas. Para permitir que isto seja feito, a função de tabela pode ser declarada como retornando o pseudotipo record. Quando este tipo de função é utilizada em uma consulta, a estrutura esperada para a linha deve ser especificada na própria consulta, para que o sistema possa saber como analisar e planejar a consulta. Considere o seguinte exemplo:

```
SELECT *
FROM dblink('dbname=meu_bd', 'select proname, prosrc from pg_proc')
AS t1(proname name, prosrc text)
WHERE proname LIKE 'bytea%';
```

A função `dblink` executa uma consulta remota (consulte `contrib/dblink`). É declarada como retornando `record`, uma vez que pode ser utilizada em qualquer tipo de consulta. O conjunto real de colunas deve ser especificado na consulta fazendo a chamada, para que o analisador saiba, por exemplo, como expandir o `*`.

## 7.2.2. A cláusula WHERE

A sintaxe da *Cláusula WHERE* é

```
WHERE condição_de_pesquisa
```

onde a *condição\_de\_pesquisa* é qualquer expressão de valor (consulte a Seção 4.2) que retorne um valor do tipo `boolean`.

Após o processamento da cláusula `FROM` ter sido feito, cada linha da tabela virtual derivada é verificada com relação à condição de pesquisa. Se o resultado da condição for verdade, a linha é mantida na tabela de saída, senão (ou seja, se o resultado for falso ou nulo) a linha é desprezada. Normalmente a condição de pesquisa faz referência a pelo menos uma coluna da tabela gerada pela cláusula `FROM`; embora isto não seja requerido, se não for assim a cláusula `WHERE` não terá utilidade.

**Nota:** A condição de junção de uma junção interna pode ser escrita tanto na cláusula `WHERE` quanto na cláusula `JOIN`. Por exemplo, estas duas expressões de tabela são equivalentes:

```
FROM a, b WHERE a.id = b.id AND b.val > 5
```

e

```
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
```

ou talvez até mesmo

```
FROM a NATURAL JOIN b WHERE b.val > 5
```

Qual destas formas deve ser utilizada é principalmente uma questão de estilo. A sintaxe do `JOIN` na cláusula `FROM` provavelmente não é muito portátil para outros sistemas gerenciadores de banco de dados SQL. Para as junções externas não existe escolha em nenhum caso: devem ser feitas na cláusula `FROM`. A cláusula `ON/USING` da junção externa *não* é equivalente à condição `WHERE`, porque determina a adição de linhas (para as linhas de entrada sem correspondência) assim como a remoção de linhas do resultado final.

Abaixo estão mostrados alguns exemplos de cláusulas `WHERE`:

```
SELECT ... FROM fdt WHERE c1 > 5
```

```
SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)
```

```
SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100
```

```
SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

sendo que `fdt` é a tabela derivada da cláusula `FROM`. As linhas que não aderem à condição de pesquisa da cláusula `WHERE` são eliminadas de `fdt`. Deve ser observada a utilização de subconsultas escalares como expressões de valor. Assim como qualquer outra consulta, as subconsultas podem utilizar expressões de tabela complexas. Deve ser observado, também, como `fdt` é referenciada nas subconsultas. A qualificação de `c1` como `fdt.c1` somente é necessária se `c1` também for o nome de uma coluna na tabela de entrada derivada da subconsulta. Entretanto, a qualificação do nome da coluna torna mais

clara a consulta, mesmo quando não é necessária. Este exemplo mostra como o escopo do nome da coluna de uma consulta externa se estende às suas consultas internas.

### 7.2.3. As cláusulas GROUP BY e HAVING

Após passar pelo filtro WHERE, a tabela de entrada derivada pode estar sujeita ao agrupamento, utilizando a cláusula GROUP BY, e à eliminação de grupos de linhas, utilizando a cláusula HAVING.

```
SELECT lista_de_seleção
      FROM ...
      [WHERE ...]
      GROUP BY referência_a_coluna_de_agrupamento [, referência_a_coluna_de_agrupamento]...
```

A Cláusula *GROUP BY* é utilizada para agrupar linhas da tabela que compartilham os mesmos valores em todas as colunas da lista. Em que ordem as colunas são listadas não faz diferença. O efeito é combinar cada conjunto de linhas que compartilham valores comuns em uma linha de grupo que representa todas as linhas do grupo. Isto é feito para eliminar redundância na saída, e/ou para calcular agregações aplicáveis a estes grupos. Por exemplo:

```
=> SELECT * FROM testel;
```

```
x | y
---+---
a | 3
c | 2
b | 5
a | 1
(4 linhas)
```

```
=> SELECT x FROM testel GROUP BY x;
```

```
x
---
a
b
c
(3 linhas)
```

Na segunda consulta não poderia ser escrito `SELECT * FROM testel GROUP BY x`, porque não existe um valor único da coluna `y` que poderia ser associado com cada grupo. As colunas agrupadas podem ser referenciadas na lista de seleção, desde que possuam um valor único em cada grupo.

De modo geral, se uma tabela for agrupada as colunas que não são usadas nos agrupamentos não podem ser referenciadas, exceto nas expressões de agregação. Um exemplo de expressão de agregação é:

```
=> SELECT x, sum(y) FROM testel GROUP BY x;
```

```
x | sum
---+---
a | 4
b | 5
c | 2
(3 linhas)
```

Aqui `sum()` é a função de agregação que calcula um valor único para o grupo todo. Mais informações sobre as funções de agregação disponíveis podem ser encontradas na Seção 9.15.

**Dica:** Um agrupamento sem expressão de agregação computa, efetivamente, o conjunto de valores distintas na coluna. Também poderia ser obtido por meio da cláusula `DISTINCT` (consulte a Seção 7.3.3).

Abaixo está mostrado um outro exemplo: cálculo do total das vendas de cada produto (e não o total das vendas de todos os produtos).

```
SELECT cod_prod, p.nome, (sum(v.unidades) * p.preco) AS vendas
FROM produtos p LEFT JOIN vendas v USING (cod_prod)
GROUP BY cod_prod, p.nome, p.preco;
```

Neste exemplo, as colunas `cod_prod`, `p.nome` e `p.preco` devem estar na cláusula `GROUP BY`, porque são referenciadas na lista de seleção da consulta (dependendo da forma exata como a tabela `produtos` for definida, as colunas `nome` e `preço` podem ser totalmente dependentes da coluna `cod_prod`, tornando os agrupamentos adicionais teoricamente desnecessários, mas isto ainda não está implementado). A coluna `v.unidades` não precisa estar na lista do `GROUP BY`, porque é usada apenas na expressão de agregação (`sum(...)`), que representa as vendas do produto. Para cada produto, a consulta retorna uma linha sumarizando todas as vendas do produto.

No SQL estrito, a cláusula `GROUP BY` somente pode agrupar pelas colunas da tabela de origem, mas o PostgreSQL estende esta funcionalidade para permitir o `GROUP BY` agrupar pelas colunas da lista de seleção. O agrupamento por expressões de valor, em vez de nomes simples de colunas, também é permitido.

Se uma tabela for agrupada utilizando a cláusula `GROUP BY`, mas houver interesse em alguns grupos apenas, pode ser utilizada a cláusula `HAVING`, de forma parecida com a cláusula `WHERE`, para eliminar grupos da tabela agrupada. A sintaxe é:

```
SELECT lista_de_seleção FROM ... [WHERE ...] GROUP BY ... HAVING expressão_booleana
```

As expressões na cláusula `HAVING` podem fazer referência tanto a expressões agrupadas quanto a não agrupadas (as quais necessariamente envolvem uma função de agregação).

Exemplo:

```
=> SELECT x, sum(y) FROM testel GROUP BY x HAVING sum(y) > 3;
```

```
x | sum
---+-----
a | 4
b | 5
(2 linhas)
```

```
=> SELECT x, sum(y) FROM testel GROUP BY x HAVING x < 'c';
```

```
x | sum
---+-----
a | 4
b | 5
(2 linhas)
```

Agora vamos fazer um exemplo mais próximo da realidade:

```
SELECT cod_prod, p.nome, (sum(v.unidades) * (p.preco - p.custo)) AS lucro
FROM produtos p LEFT JOIN vendas v USING (cod_prod)
WHERE v.data > CURRENT_DATE - INTERVAL '4 weeks'
GROUP BY cod_prod, p.nome, p.preco, p.custo
HAVING sum(p.preco * v.unidades) > 5000;
```

No exemplo acima, a cláusula `WHERE` está selecionando linhas por uma coluna que não é agrupada (a expressão somente é verdadeira para as vendas feitas nas quatro últimas semanas, enquanto a cláusula `HAVING` restringe a saída aos grupos com um total de vendas brutas acima de 5000. Deve ser observado que as expressões de agregação não precisam ser necessariamente as mesmas em todas as partes da consulta.

### Exemplo 7-1. Utilização de `HAVING` sem `GROUP BY` no `SELECT`

O exemplo abaixo mostra a utilização da cláusula `HAVING` sem a cláusula `GROUP BY` no comando `SELECT`. É criada a tabela `produtos` e são inseridas cinco linhas. Quando a cláusula `HAVING` exige a presença de mais de cinco linhas na tabela, a consulta não retorna nenhuma linha.<sup>3 4</sup>

```
=> create global temporary table produtos(codigo int, valor float);
=> insert into produtos values (1, 102);
=> insert into produtos values (2, 104);
```

```
=> insert into produtos values (3, 202);
=> insert into produtos values (4, 203);
=> insert into produtos values (5, 204);

=> select avg(valor) from produtos;

      avg
-----
      163
(1 linha)

=> select avg(valor) from produtos having count(*)>=5;

      avg
-----
      163
(1 linha)

=> select avg(valor) from produtos having count(*)=5;

      avg
-----
      163
(1 linha)

=> select avg(valor) from produtos having count(*)>5;

      avg
-----
      163
(1 linha)

=> select avg(valor) from produtos having count(*)>5;

      avg
-----
(0 linhas)
```

### Exemplo 7-2. Utilização da expressão CASE para agrupar valores

A expressão CASE pode fazer parte da lista de agrupamento. Este exemplo usa a expressão CASE para agrupar as notas dos alunos em conceitos, e calcular a nota mínima, máxima e média, além da quantidade de notas, correspondente a cada conceito. Abaixo está mostrado o script utilizado: <sup>5 6</sup>

```
CREATE TABLE notas (
    nota decimal(4,2) CONSTRAINT chknota
        CHECK (nota BETWEEN 0.00 AND 10.00)
);
INSERT INTO notas VALUES(10);
INSERT INTO notas VALUES(9.2);
INSERT INTO notas VALUES(9.0);
INSERT INTO notas VALUES(8.3);
INSERT INTO notas VALUES(7.7);
INSERT INTO notas VALUES(7.4);
INSERT INTO notas VALUES(6.4);
INSERT INTO notas VALUES(5.8);
INSERT INTO notas VALUES(5.1);
INSERT INTO notas VALUES(5.0);
INSERT INTO notas VALUES(0);
SELECT CASE
    WHEN nota < 3 THEN 'E'
    WHEN nota < 5 THEN 'D'
    WHEN nota < 7 THEN 'C'
    WHEN nota < 9 THEN 'B'
    ELSE 'A'
END AS conceito,
COUNT(*) AS quantidade,
MIN(nota) AS menor,
MAX(nota) AS maior,
AVG(nota) AS media
```

```

FROM notas
GROUP BY CASE
    WHEN nota < 3 THEN 'E'
    WHEN nota < 5 THEN 'D'
    WHEN nota < 7 THEN 'C'
    WHEN nota < 9 THEN 'B'
    ELSE 'A'
END
ORDER BY conceito;

```

A seguir estão mostrados os resultados obtidos:

conceito	quantidade	menor	maior	media
A	3	9.00	10.00	9.4000000000000000
B	3	7.40	8.30	7.8000000000000000
C	4	5.00	6.40	5.5750000000000000
E	1	0.00	0.00	0.0000000000000000

(4 linhas)

### Exemplo 7-3. Utilização da expressão CASE em chamada de função

A expressão CASE pode ser usada como argumento de chamada de função. Este exemplo usa a expressão CASE como argumento da função COUNT, passando o valor 1 quando a nota corresponde ao conceito, e nulo quando não corresponde. Desta forma, a função COUNT conta a quantidade de notas presentes em cada conceito, uma vez que os valores nulos não são contados. Os dados são os mesmos do exemplo anterior. Abaixo está mostrada a consulta utilizada:<sup>7 8</sup>

```

SELECT COUNT(CASE WHEN nota BETWEEN 9.00 AND 10.00 THEN 1 ELSE NULL END) AS A,
       COUNT(CASE WHEN nota BETWEEN 7.00 AND 8.99 THEN 1 ELSE NULL END) AS B,
       COUNT(CASE WHEN nota BETWEEN 5.00 AND 6.99 THEN 1 ELSE NULL END) AS C,
       COUNT(CASE WHEN nota BETWEEN 3.00 AND 4.99 THEN 1 ELSE NULL END) AS D,
       COUNT(CASE WHEN nota BETWEEN 0.00 AND 2.99 THEN 1 ELSE NULL END) AS E
FROM notas;

```

A seguir estão mostrados os resultados obtidos:

a	b	c	d	e
3	3	4	0	1

(1 linha)

Desta forma, foi mostrado em uma linha o mesmo resultado da coluna quantidade do exemplo anterior.

### Exemplo 7-4. Combinação de informação agrupada e não agrupada

Os comandos SELECT que retornam apenas uma linha<sup>9</sup> podem ser utilizados para combinar informações agrupadas com informações não agrupadas na mesma consulta. Neste exemplo cada nota é mostrada junto com a menor nota, a maior nota, e a média de todas as notas. Os dados são os mesmos dos dois exemplos anteriores. Abaixo está mostrada a consulta utilizada:<sup>10 11</sup>

```

SELECT nota,
       (SELECT MIN(nota) FROM notas) AS menor,
       (SELECT MAX(nota) FROM notas) AS maior,
       (SELECT AVG(nota) FROM notas) AS media
FROM notas;

```

A seguir estão mostrados os resultados obtidos:

nota	menor	maior	media
10.00	0.00	10.00	6.7181818181818182
9.20	0.00	10.00	6.7181818181818182
9.00	0.00	10.00	6.7181818181818182
8.30	0.00	10.00	6.7181818181818182
7.70	0.00	10.00	6.7181818181818182
7.40	0.00	10.00	6.7181818181818182
6.40	0.00	10.00	6.7181818181818182
5.80	0.00	10.00	6.7181818181818182



```
5.10 | 0.00 | 10.00 | 6.7181818181818182
5.00 | 0.00 | 10.00 | 6.7181818181818182
0.00 | 0.00 | 10.00 | 6.7181818181818182
(11 linhas)
```

## 7.3. Listas de seleção

Conforme foi mostrado na seção anterior, a expressão de tabela do comando `SELECT` constrói uma tabela virtual intermediária, possivelmente por meio da combinação de tabelas, visões, eliminação de linhas, agrupamento, etc. Esta tabela é finalmente passada adiante para ser processada pela *lista de seleção*. A lista de seleção determina quais *colunas* da tabela intermediária vão realmente para a saída.

### 7.3.1. Itens da lista de seleção

O tipo mais simples de lista de seleção é o `*`, que emite todas as colunas produzidas pela expressão de tabela. De outra forma, a lista de seleção é uma lista separada por vírgulas de expressões de valor (conforme definido na Seção 4.2). Por exemplo, esta pode ser uma lista de nomes de colunas:

```
SELECT a, b, c FROM ...
```

Os nomes das colunas `a`, `b` e `c` podem ser os nomes verdadeiros das colunas das tabelas referenciadas na cláusula `FROM`, ou aliases dados a estas colunas conforme explicado na Seção 7.2.1.2. O espaço de nomes disponível na lista de seleção é o mesmo da cláusula `WHERE`, a não ser que seja utilizado agrupamento e, neste caso, passa a ser o mesmo da cláusula `HAVING`.

Quando mais de uma tabela possui uma coluna com o mesmo nome, o nome da tabela deve ser fornecido também, como em:

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

Ao se trabalhar com várias tabelas, também pode ser útil solicitar todas as colunas de uma determinada tabela:

```
SELECT tbl1.*, tbl2.a FROM ...
```

(Consulte também a Seção 7.2.2)

Se for utilizada uma expressão de valor arbitrária na lista de seleção, esta expressão adiciona, conceitualmente, uma nova coluna virtual à tabela retornada. A expressão de valor é avaliada uma vez para cada linha do resultado, com os valores da linha substituídos nas referências a coluna. Porém, as expressões da lista de seleção não precisam referenciar nenhuma coluna da expressão de tabela da cláusula `FROM`; podem ser, inclusive, expressões aritméticas constantes, por exemplo.

### 7.3.2. Rótulos de coluna

Podem ser atribuídos nomes para as entradas da lista de seleção para processamento posterior. Neste caso, “processamento posterior” é uma especificação opcional de classificação e o aplicativo cliente (por exemplo, os títulos das colunas para exibição). Por exemplo:

```
SELECT a AS valor, b + c AS soma FROM ...
```

Se nenhum nome de coluna de saída for especificado utilizando `AS`, o sistema atribui um nome padrão. Para referências a colunas simples, é o nome da coluna referenciada. Para chamadas de função, é o nome da função. Para expressões complexas o sistema gera um nome genérico.

**Nota:** Aqui, o nome dado à coluna de saída é diferente do nome dado na cláusula `FROM` (consulte a Seção 7.2.1.2). Na verdade, este processo permite mudar o nome da mesma coluna duas vezes, mas o nome escolhido na lista de seleção é o passado adiante.

### 7.3.3. DISTINCT

Após a lista de seleção ser processada, a tabela resultante pode opcionalmente estar sujeita à remoção das linhas duplicadas. A palavra chave `DISTINCT` deve ser escrita logo após o `SELECT` para especificar esta funcionalidade:

```
SELECT DISTINCT lista_de_seleção ...
```

(Em vez de `DISTINCT` pode ser utilizada a palavra `ALL` para especificar o comportamento padrão de manter todas as linhas)

Como é óbvio, duas linhas são consideradas distintas quando têm pelo menos uma coluna diferente. Os valores nulos são considerados iguais nesta comparação.

Como alternativa, uma expressão arbitrária pode determinar quais linhas devem ser consideradas distintas:

```
SELECT DISTINCT ON (expressão [, expressão ...]) lista_de_seleção ...
```

Neste caso, *expressão* é uma expressão de valor arbitrária avaliada para todas as linhas. Um conjunto de linhas para as quais todas as expressões são iguais são consideradas duplicadas, e somente a primeira linha do conjunto é mantida na saída. Deve ser observado que a “primeira linha” de um conjunto é imprevisível, a não ser que a consulta seja ordenada por um número suficiente de colunas para garantir a ordem única das linhas que chegam no filtro `DISTINCT` (o processamento de `DISTINCT ON` ocorre após a ordenação do `ORDER BY`).

A cláusula `DISTINCT ON` não faz parte do padrão SQL, sendo algumas vezes considerada um estilo ruim devido à natureza potencialmente indeterminada de seus resultados. Utilizando-se adequadamente `GROUP BY` e subconsultas no `FROM` esta construção pode ser evitada, mas geralmente é a alternativa mais fácil.

## 7.4. Combinação de consultas

Pode-se combinar os resultados de duas consultas utilizando as operações de conjunto união, interseção e diferença<sup>12 13 14 15</sup>. A sintaxe é

```
consulta1 UNION [ALL] consulta2
consulta1 INTERSECT [ALL] consulta2
consulta1 EXCEPT [ALL] consulta2
```

onde *consulta1* e *consulta2* são consultas que podem utilizar qualquer uma das funcionalidades mostradas até aqui. As operações de conjuntos também podem ser aninhadas ou encadeadas. Por exemplo:

```
consulta1 UNION consulta2 UNION consulta3
```

significa, na verdade,

```
(consulta1 UNION consulta2) UNION consulta3
```

Efetivamente, `UNION` anexa o resultado da *consulta2* ao resultado da *consulta1* (embora não haja garantia que esta seja a ordem que as linhas realmente retornam). Além disso, são eliminadas do resultado as linhas duplicadas, do mesmo modo que no `DISTINCT`, a não ser que seja utilizado `UNION ALL`.

`INTERSECT` retorna todas as linhas presentes tanto no resultado da *consulta1* quanto no resultado da *consulta2*. As linhas duplicadas são eliminadas, a não ser que seja utilizado `INTERSECT ALL`.

`EXCEPT` retorna todas as linhas presentes no resultado da *consulta1*, mas que não estão presentes no resultado da *consulta2* (às vezes isto é chamado de *diferença* entre duas consultas). Novamente, as linhas duplicadas são eliminadas a não ser que seja utilizado `EXCEPT ALL`.

Para ser possível calcular a união, a interseção, ou a diferença entre duas consultas, as duas consultas devem ser “compatíveis para união”, significando que ambas devem retornar o mesmo número de colunas, e que as colunas correspondentes devem possuir tipos de dado compatíveis, conforme descrito na Seção 10.5.

**Nota:** O exemplo abaixo foi escrito pelo tradutor, não fazendo parte do manual original.

### Exemplo 7-5. Linhas diferentes em duas tabelas com definições idênticas

Este exemplo mostra a utilização de `EXCEPT` e `UNION` para descobrir as linhas diferentes de duas tabelas semelhantes.

```
CREATE TEMPORARY TABLE a (c1 text, c2 text, c3 text);
INSERT INTO a VALUES ('x', 'x', 'x');
INSERT INTO a VALUES ('x', 'x', 'y'); -- nas duas tabelas
INSERT INTO a VALUES ('x', 'y', 'x');
```

```
CREATE TEMPORARY TABLE b (c1 text, c2 text, c3 text);
INSERT INTO b VALUES ('x', 'x', 'y'); -- nas duas tabelas
INSERT INTO b VALUES ('x', 'x', 'y'); -- nas duas tabelas
INSERT INTO b VALUES ('x', 'y', 'y');
INSERT INTO b VALUES ('y', 'y', 'y');
INSERT INTO b VALUES ('y', 'y', 'y');
```

-- No comando abaixo só um par ('x', 'x', 'y') é removido do resultado  
 -- Este comando executa no DB2 8.1 sem alterações.

```
(SELECT 'a-b' AS dif, a.* FROM a EXCEPT ALL SELECT 'a-b', b.* FROM b)
UNION ALL
(SELECT 'b-a', b.* FROM b EXCEPT ALL SELECT 'b-a', a.* FROM a);
```

```

dif | c1 | c2 | c3
-----+-----+-----+-----
a-b | x  | x  | x
a-b | x  | y  | x
b-a | x  | x  | y
b-a | x  | y  | y
b-a | y  | y  | y
b-a | y  | y  | y
(6 linhas)
```

-- No comando abaixo são removidas todas as linhas ('x', 'x', 'y'),  
 -- e só é mostrada uma linha ('y', 'y', 'y') no resultado  
 -- Este comando executa no DB2 8.1 sem alterações.  
 -- Este comando executa no Oracle 10g trocando EXCEPT por MINUS.

```
(SELECT 'a-b' AS dif, a.* FROM a EXCEPT SELECT 'a-b', b.* FROM b)
UNION
(SELECT 'b-a', b.* FROM b EXCEPT SELECT 'b-a', a.* FROM a);
```

```

dif | c1 | c2 | c3
-----+-----+-----+-----
a-b | x  | x  | x
a-b | x  | y  | x
b-a | x  | y  | y
b-a | y  | y  | y
(4 linhas)
```

## 7.5. Ordenação de linhas

Após a consulta ter produzido a tabela de saída (após a lista de seleção ter sido processada) esta tabela pode, opcionalmente, ser ordenada. Se nenhuma ordenação for especificada, as linhas retornam em uma ordem aleatória. Neste caso, a ordem real depende dos tipos de plano de varredura e de junção e da ordem no disco, mas não se deve confiar nisto. Uma ordem de saída específica somente pode ser garantida se a etapa de ordenação for especificada explicitamente.

A cláusula `ORDER BY` especifica a ordem de classificação:

```
SELECT lista_de_seleção
      FROM expressão_de_tabela
      ORDER BY coluna1 [ASC | DESC] [, coluna2 [ASC | DESC] ...]
```

onde *coluna1*, etc., fazem referência às colunas da lista de seleção. Pode ser tanto o nome de saída da coluna (consulte a Seção 7.3.2) quanto o número da coluna. Alguns exemplos:

```
SELECT a, b FROM tabela1 ORDER BY a;
SELECT a + b AS soma, c FROM tabela1 ORDER BY soma;
SELECT a, sum(b) FROM tabela1 GROUP BY a ORDER BY 1;
```

Como extensão ao padrão SQL, o PostgreSQL também permite ordenar por expressões arbitrárias:

```
SELECT a, b FROM tabela1 ORDER BY a + b;
```

Também é permitido fazer referência a nomes de colunas da cláusula `FROM` que não estão presentes na lista de seleção:

```
SELECT a FROM tabela1 ORDER BY b;
```

Mas estas extensões não funcionam nas consultas envolvendo `UNION`, `INTERSECT` ou `EXCEPT`, e não são portáteis para outros bancos de dados SQL.

Cada especificação de coluna pode ser seguida pela palavra opcional `ASC` ou `DESC`, para definir a direção de ordenação como ascendente ou decendente. A ordem `ASC` é o padrão. A ordenação ascendente coloca os valores menores na frente, sendo que “menor” é definido nos termos do operador `<`. De forma semelhante, a ordenação decendente é determinada pelo operador `>`.<sup>16</sup>

Se for especificada a ordenação por mais de uma coluna, as últimas entradas são utilizadas para ordenar as linhas iguais sob a ordem imposta pelas colunas de ordenação anteriores.

## 7.6. LIMIT e OFFSET

`LIMIT` (limite) e `OFFSET` (deslocamento) permitem que seja trazida apenas uma parte das linhas geradas pelo restante da consulta:

```
SELECT lista_de_seleção
      FROM expressão_de_tabela
      [LIMIT { número | ALL }] [OFFSET número]
```

Se for especificado o limite, não será retornada mais que esta quantidade de linhas (mas possivelmente menos, se a consulta produzir menos linhas). `LIMIT ALL` é o mesmo que omitir a cláusula `LIMIT`.

`OFFSET` diz para saltar esta quantidade de linhas antes de começar a retornar as linhas. `OFFSET 0` é o mesmo que omitir a cláusula `OFFSET`. Se forem especificados tanto `OFFSET` quanto `LIMIT`, então são saltadas `OFFSET` linhas antes de começar a contar as `LIMIT` linhas que serão retornadas.

Quando se utiliza `LIMIT` é importante utilizar a cláusula `ORDER BY` para estabelecer uma ordem única para as linhas do resultado. Caso contrário, será retornado um subconjunto imprevisível de linhas da consulta; pode-se desejar obter da décima a vigésima linha, mas da décima a vigésima de qual ordem? A ordem é desconhecida a não ser que seja especificado `ORDER BY`.

O otimizador de consultas leva `LIMIT` em consideração para gerar o plano da consulta, portanto é bastante provável obter planos diferentes (resultando em uma ordem diferente das linhas) dependendo do que for especificado para `LIMIT` e `OFFSET`. Portanto, utilizar valores diferentes de `LIMIT/OFFSET` para selecionar subconjuntos diferentes do resultado da consulta *produz resultados inconsistentes*, a não ser que seja imposta uma ordem previsível do resultado por meio da cláusula `ORDER BY`. Isto não está errado; isto é uma consequência inerente ao fato do SQL não prometer retornar os resultados de uma consulta em qualquer ordem específica, a não ser que `ORDER BY` seja utilizado para impor esta ordem.

É necessário computar as linhas saltadas pelo `OFFSET` no servidor; portanto, um `OFFSET` grande pode ser ineficiente.

## Notas

1. Tabela escrita pelo tradutor, não fazendo parte do manual original.
2. SQL Server — aliás: um nome alternativo para tabela ou coluna em expressões, geralmente utilizado para encurtar o nome em uma referência subsequente no código, evitar possíveis referências ambíguas, ou fornecer um nome mais descritivo para a saída do comando. Um aliás também pode ser um nome alternativo para o servidor. SQL Server Books Online (N. do T.)
3. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
4. Oracle — Este exemplo foi executado no Oracle 10g e produziu os mesmos resultados.
5. Exemplo escrito pelo tradutor, não fazendo parte do manual original, baseado no exemplo do livro DB2® Universal Database V8 for Linux, UNIX, and Windows Database Administration Certification Guide, 5th Edition

(<http://www.phptr.com/title/0130463612>), George Baklarz e Bill Wong, Series IBM Press, Prentice Hall Professional Technical Reference, 2003, pág. 375.

6. Este exemplo foi executado no SQL Server 2000, no Oracle 10g e no DB2 8.1 sem alterações, produzindo o mesmo resultado.
7. Exemplo escrito pelo tradutor, não fazendo parte do manual original, baseado em exemplo do mesmo livro do exemplo anterior, pág. 376.
8. Esta consulta foi executada no SQL Server 2000, no Oracle 10g e no DB2 8.1 sem alterações, produzindo o mesmo resultado.
9. Scalar-fullselect  
(<http://publib.boulder.ibm.com/infocenter/dzichelp/index.jsp?topic=/com.ibm.db2.doc.sqlref/bjnrmsr158.htm>) — É um comando `SELECT` completo entre parênteses, que retorna uma única linha contendo um único valor de coluna. Se não retornar nenhuma linha o resultado da expressão é o valor nulo, e se retornar mais de uma linha ocorre um erro.
10. Exemplo escrito pelo tradutor, não fazendo parte do manual original, baseado em exemplo do mesmo livro do exemplo anterior, pág. 316.
11. Esta consulta foi executada no SQL Server 2000, no Oracle 10g e no DB2 8.1 sem alterações, produzindo o mesmo resultado.
12. Dados dois conjuntos A e B: chama-se *diferença* entre A e B o conjunto formado pelos elementos de A que não pertencem a B; chama-se *interseção* de A com B o conjunto formado pelos elementos comuns ao conjunto A e ao conjunto B; chama-se *união* de A com B o conjunto formado pelos elementos que pertencem a A ou B. Edwaldo Bianchini e Herval Paccola - Matemática - Operações com conjuntos. (N. do T.)
13. SQL Server 2000 — possui o operador `UNION [ALL]`, mas não possui os operadores `INTERSECT` e `EXCEPT`, embora estas duas sejam palavras reservadas. (N. do T.)
14. Oracle 9i — possui os operadores `UNION [ALL]`, `INTERSECT` e `MINUS` (equivalente ao `EXCEPT`). Set Operators (<http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/operators5.htm>). (N. do T.)
15. DB2 8.1 — possui os operadores `UNION [ALL]`, `INTERSECT [ALL]` e `EXCEPT [ALL]`. (N. do T.)
16. Na verdade, o PostgreSQL utiliza a *classe de operadores B-tree padrão* para o tipo de dado da coluna para determinar a ordem de classificação para `ASC` e `DESC`. Por convenção, os tipos de dado são configurados de maneira que os operadores `<` e `>` correspondam a esta ordem de classificação, mas o projetista de um tipo de dado definido pelo usuário pode decidir fazer algo diferente.

## Capítulo 8. Tipos de dado

O PostgreSQL disponibiliza para os usuários um amplo conjunto de tipos de dado nativos. Os usuários podem adicionar novos tipos ao PostgreSQL utilizando o comando `CREATE TYPE`.<sup>1 2 3</sup>

A Tabela 8-1 mostra todos os tipos de dado nativos de propósito geral. A maioria dos nomes alternativos listados na coluna “Aliases” é o nome utilizado internamente pelo PostgreSQL por motivos históricos. Além desses, existem alguns tipos usados internamente ou em obsolescência<sup>4</sup> que não são mostrados aqui.

**Tabela 8-1. Tipos de dado**

Nome	Aliases	Descrição
<code>bigint</code>	<code>int8</code>	inteiro de oito bytes com sinal <sup>a</sup>
<code>bigserial</code>	<code>serial8</code>	inteiro de oito bytes com auto-incremento
<code>bit [ (n) ]</code>		cadeia de bits de comprimento fixo
<code>bit varying [ (n) ]</code>	<code>varbit</code>	cadeia de bits de comprimento variável <sup>b</sup>
<code>boolean</code>	<code>bool</code>	booleano lógico (verdade/falso)
<code>box</code>		caixa retangular no plano
<code>bytea</code>		dados binários (“matriz de bytes”)
<code>character varying [ (n) ]</code>	<code>varchar [ (n) ]</code>	cadeia de caracteres de comprimento variável <sup>c</sup>
<code>character [ (n) ]</code>	<code>char [ (n) ]</code>	cadeia de caracteres de comprimento fixo
<code>cidr</code>		endereço de rede IPv4 ou IPv6
<code>circle</code>		círculo no plano
<code>date</code>		data de calendário (ano, mês, dia)
<code>double precision</code>	<code>float8</code>	número de ponto flutuante de precisão dupla <sup>d</sup>
<code>inet</code>		endereço de hospedeiro IPv4 ou IPv6
<code>integer</code>	<code>int</code> , <code>int4</code>	inteiro de quatro bytes com sinal
<code>interval [ (p) ]</code>		espaço de tempo
<code>line</code>		linha infinita no plano
<code>lseg</code>		segmento de linha no plano
<code>macaddr</code>		endereço MAC
<code>money</code>		quantia monetária
<code>numeric [ (p, s) ]</code>	<code>decimal [ (p, s) ]</code>	numérico exato com precisão selecionável
<code>path</code>		caminho geométrico no plano
<code>point</code>		ponto geométrico no plano
<code>polygon</code>		caminho geométrico fechado no plano
<code>real</code>	<code>float4</code>	número de ponto flutuante de precisão simples

Nome	Aliases	Descrição
<code>smallint</code>	<code>int2</code>	inteiro de dois bytes com sinal
<code>serial</code>	<code>serial4</code>	inteiro de quatro bytes com auto-incremento
<code>text</code>		cadeia de caracteres de comprimento variável
<code>time [ (p) ] [ without time zone ]</code>		hora do dia
<code>time [ (p) ] with time zone</code>	<code>timetz</code>	hora do dia, incluindo a zona horária
<code>timestamp [ (p) ] [ without time zone ]</code>		data e hora
<code>timestamp [ (p) ] with time zone</code>	<code>timestampz</code>	data e hora, incluindo a zona horária
<p>Notas:</p> <p>a. Os tipos de dado NUMERIC, DECIMAL, SMALLINT, INTEGER e BIGINT são referenciados coletivamente como <i>tipos numéricos exatos</i>. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)</p> <p>b. comprimento variável — uma característica das cadeias de caracteres e das cadeias binárias que permite as cadeias conterem qualquer número de caracteres ou de octetos, respectivamente, entre zero e um número máximo, conhecido como comprimento máximo em caracteres ou octetos, respectivamente, da cadeia. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)</p> <p>c. Os tipos de dado CHARACTER, CHARACTER VARYING e CHARACTER LARGE OBJECT são referenciados coletivamente como <i>tipos cadeia de caracteres</i>. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)</p> <p>d. Os tipos de dado FLOAT, REAL e DOUBLE PRECISION são referenciados coletivamente como <i>tipos numéricos aproximados</i>. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)</p>		

**Compatibilidade:** Os seguintes tipos (ou a citação destes) são especificados pelo padrão SQL: `bit`, `bit varying`, `boolean`, `char`, `character varying`, `character`, `varchar`, `date`, `double precision`, `integer`, `interval`, `numeric`, `decimal`, `real`, `smallint`, `time` (com ou sem zona horária), `timestamp` (com ou sem zona horária).

Cada tipo de dado possui uma representação externa determinada pelas suas funções de entrada e de saída. Muitos tipos nativos possuem formato externo óbvio. Entretanto, muitos tipos existem apenas no PostgreSQL, como os caminhos geométricos, ou possuem várias possibilidades para o formato, como os tipos de data e hora. Algumas das funções de entrada e saída não são inversíveis, ou seja, o resultado da função de saída pode perder precisão quando comparado com a entrada original.

## 8.1. Tipos numéricos

Os tipos numéricos consistem em inteiros de dois, quatro e oito bytes, números de ponto flutuante de quatro e oito bytes, e decimais de precisão selecionável. A Tabela 8-2 lista os tipos disponíveis.

**Tabela 8-2. Tipos numéricos**

Nome	Tamanho de armazenamento	Descrição	Faixa de valores
<code>smallint</code>	2 bytes	inteiro com faixa pequena	-32768 a +32767
<code>integer</code>	4 bytes	escolha usual para inteiro	-2147483648 a +2147483647

Nome	Tamanho de armazenamento	Descrição	Faixa de valores
<code>bigint</code>	8 bytes	inteiro com faixa larga	-9223372036854775808 a 9223372036854775807
<code>decimal</code>	variável	precisão especificada pelo usuário, exato	sem limite
<code>numeric</code>	variável	precisão especificada pelo usuário, exato	sem limite
<code>real</code>	4 bytes	precisão variável, inexato	precisão de 6 dígitos decimais
<code>double precision</code>	8 bytes	precisão variável, inexato	precisão de 15 dígitos decimais
<code>serial</code>	4 bytes	inteiro com auto-incremento	1 a 2147483647
<code>bigserial</code>	8 bytes	inteiro grande com auto-incremento	1 a 9223372036854775807

A sintaxe das constantes para os tipos numéricos é descrita na Seção 4.1.2. Os tipos numéricos possuem um conjunto completo de operadores aritméticos e funções correspondentes. Consulte o Capítulo 9 para obter informações adicionais. As próximas seções descrevem os tipos em detalhe.

### 8.1.1. Tipos inteiros

Os tipos `smallint`, `integer` e `bigint` armazenam números inteiros, ou seja, números sem a parte fracionária, com faixas diferentes. A tentativa de armazenar um valor fora da faixa permitida resulta em erro.

O tipo `integer` é a escolha usual, porque oferece o melhor equilíbrio entre faixa de valores, tamanho de armazenamento e desempenho. Geralmente o tipo `smallint` só é utilizado quando o espaço em disco está muito escasso. O tipo `bigint` somente deve ser usado quando a faixa de valores de `integer` não for suficiente, porque este último é bem mais rápido.

O tipo `bigint` pode não funcionar de modo correto em todas as plataformas, porque depende de suporte no compilador para inteiros de oito bytes. Nas máquinas sem este suporte, o `bigint` age do mesmo modo que o `integer` (mas ainda ocupa oito bytes de armazenamento). Entretanto, não é de nosso conhecimento nenhuma plataforma razoável onde este caso se aplique.

O padrão SQL somente especifica os tipos inteiros `integer` (ou `int`) e `smallint`. O tipo `bigint`, e os nomes de tipo `int2`, `int4` e `int8` são extensões, também compartilhadas por vários outros sistemas de banco de dados SQL.

### 8.1.2. Números com precisão arbitrária

O tipo `numeric` pode armazenar números com precisão de até 1.000 dígitos e realizar cálculos exatos. É recomendado, especialmente, para armazenar quantias monetárias e outras quantidades onde se requeira exatidão. Entretanto, a aritmética em valores do tipo `numeric` é muito lenta se comparada com os tipos inteiros, ou com os tipos de ponto flutuante descritos na próxima seção.

São utilizados os seguintes termos: A *escala* do tipo `numeric` é o número de dígitos decimais da parte fracionária, à direita do ponto decimal. A *precisão* do tipo `numeric` é o número total de dígitos significativos de todo o número, ou seja, o número de dígitos nos dois lados do ponto decimal. Portanto, o número 23.5141<sup>5</sup> possui precisão igual a 6 e escala igual a 4. Os inteiros podem ser considerados como tendo escala igual a zero.

Tanto a precisão máxima quanto a escala de uma coluna do tipo `numeric` podem ser configuradas. Para declarar uma coluna do tipo `numeric` é utilizada a sintaxe:

```
NUMERIC(precisão, escala)
```

A precisão deve ser um número positivo, enquanto a escala pode ser zero ou positiva. Como forma alternativa,



`NUMERIC(precisão)`

define a escala como sendo igual a 0. Especificando-se

`NUMERIC`

sem qualquer precisão ou escala é criada uma coluna onde podem ser armazenados valores numéricos com qualquer precisão ou escala, até a precisão limite da implementação. Uma coluna deste tipo não converte os valores de entrada para nenhuma escala em particular, enquanto as colunas do tipo `numeric` com escala declarada convertem os valores da entrada para esta escala (O padrão SQL requer a escala padrão igual a 0, ou seja, uma conversão para a precisão inteira. Isto foi considerado sem utilidade. Havendo preocupação com a portabilidade, a precisão e a escala devem ser sempre especificadas explicitamente).

Se a escala do valor a ser armazenado for maior que a escala declarada para a coluna, o sistema arredonda o valor para o número de dígitos fracionários especificado. Depois, se o número de dígitos à esquerda do ponto decimal exceder a precisão declarada menos a escala declarada, é gerado um erro.

### Exemplo 8-1. Arredondamento em tipo `numeric`

Abaixo estão mostrados exemplos de inserção de dados em um campo do tipo `numeric`. No terceiro exemplo o arredondamento faz com que a precisão do campo seja excedida.<sup>6</sup>

A execução deste exemplo no SQL Server 2000 e no Oracle 10g produziu o mesmo resultado, mas o DB2 8.1 em vez de arredondar trunca as casas decimais e, por isso, a precisão não é excedida.

```
=> CREATE TABLE t ( c NUMERIC(6,3));
=> INSERT INTO t VALUES (998.9991);
=> INSERT INTO t VALUES (998.9999);
=> SELECT * FROM t;
```

```
      c
-----
998.999
999.000
(2 linhas)
```

```
=> INSERT INTO t VALUES (999.9999);
```

ERRO: estouro de campo numérico

DETALHE: O valor absoluto é maior ou igual a 10<sup>3</sup> para campo com precisão 6, escala 3.

Os valores numéricos são armazenados fisicamente sem zeros adicionais no início ou no final. Portanto, a precisão e a escala declaradas para uma coluna são as alocações máximas, e não fixas (Sob este aspecto o tipo `numeric` é mais semelhante ao tipo `varchar(n)` do que ao tipo `char(n)`).

Além dos valores numéricos ordinários o tipo `numeric` aceita o valor especial NaN, que significa “não-é-um-número” (*not-a-number*). Toda operação envolvendo NaN produz outro NaN. Para escrever este valor como uma constante em um comando SQL deve-se colocá-lo entre apóstrofes como, por exemplo, `UPDATE tabela SET x = 'NaN'`. Na entrada, a cadeia de caracteres NaN é reconhecida sem que haja distinção entre letras maiúsculas e minúsculas.

Os tipos `decimal` e `numeric` são equivalentes. Os dois tipos fazem parte do padrão SQL.

### 8.1.3. Tipos de ponto flutuante

Os tipos de dado `real` e `double precision` são tipos numéricos não exatos de precisão variável. Na prática, estes tipos são geralmente implementações do “Padrão IEEE 754 para Aritmética Binária de Ponto Flutuante” (de precisão simples e dupla, respectivamente), conforme suportado pelo processador, sistema operacional e compilador utilizados.

Não exato significa que alguns valores não podem ser convertidos exatamente para o formato interno, sendo armazenados como aproximações. Portanto, ao se armazenar e posteriormente imprimir um valor podem ocorrer pequenas discrepâncias. A gerência destes erros, e como se propagam através dos cálculos, é assunto de um ramo da matemática e da ciência da computação que não será exposto aqui, exceto os seguintes pontos:

- Se for necessário armazenamento e cálculos exatos (como em quantias monetárias), em vez de tipos de ponto flutuante deve ser utilizado o tipo `numeric`.
- Se for desejado efetuar cálculos complicados usando tipos de ponto flutuante para algo importante, especialmente dependendo de certos comportamentos em situações limites (infinito ou muito próximo de zero), a implementação deve ser avaliada cuidadosamente.
- A comparação de igualdade de dois valores de ponto flutuante pode funcionar conforme o esperado, ou não.

Na maioria das plataformas o tipo `real` possui uma faixa de pelo menos  $1\text{E}-37$  a  $1\text{E}+37$ , com precisão de pelo menos 6 dígitos decimais. O tipo `double precision` normalmente possui uma faixa em torno de  $1\text{E}-307$  a  $1\text{E}+308$  com precisão de pelo menos 15 dígitos. Os valores muito pequenos ou muito grandes causam erro. O arredondamento pode acontecer se a precisão do número entrado for muito grande. Os números muito próximos de zero, que não podem ser representados de forma distinta de zero, causam erro de `underflow`.

Além dos valores numéricos ordinários, os tipos de ponto flutuante possuem diversos valores especiais:

```
Infinity
-Infinity
NaN
```

Estes valores representam os valores especiais do padrão IEEE 754 “infinito”, “infinito negativo” e “não-é-um-número”, respectivamente (Nas máquinas cuja aritmética de ponto flutuante não segue o padrão IEEE 754, estes valores provavelmente não vão funcionar da forma esperada). Ao se escrever estes valores como constantes em um comando SQL deve-se colocá-los entre apóstrofes como, por exemplo, `UPDATE tabela SET x = 'Infinity'`. Na entrada, estas cadeias de caracteres são reconhecidas sem que haja distinção entre letras maiúsculas e minúsculas

O PostgreSQL também suporta a notação do padrão SQL `float` e `float(p)` para especificar tipos numéricos inexatos. Neste caso, *p* especifica a precisão mínima aceitável em dígitos binários. O PostgreSQL aceita de `float(1)` a `float(24)` como selecionando o tipo `real`, enquanto `float(25)` a `float(53)` selecionam `double precision`. Os valores de *p* fora da faixa permitida ocasionam erro. `float` sem precisão especificada é assumido como significando `double precision`.

**Nota:** Antes do PostgreSQL 7.4 a precisão em `float(p)` era considerada como significando a quantidade de dígitos decimais, mas foi corrigida para corresponder ao padrão SQL, que especifica que a precisão é medida em dígitos binários. A premissa que `real` e `double precision` possuem exatamente 24 e 53 bits na mantissa, respectivamente, está correta para implementações em acordo com o padrão IEEE para números de ponto flutuante. Nas plataformas não-IEEE pode ser um pouco diferente, mas para simplificar as mesmas faixas de *p* são utilizadas em todas as plataformas.

#### 8.1.4. Tipos serials

Os tipos de dado `serial` e `bigserial` não são tipos verdadeiros, mas meramente uma notação conveniente para definir colunas identificadoras únicas (semelhante à propriedade `AUTO_INCREMENTO` existente em alguns outros bancos de dados). Na implementação corrente especificar

```
CREATE TABLE nome_da_tabela (
    nome_da_coluna SERIAL
);
```

equivale a especificar:

```
CREATE SEQUENCE nome_da_tabela_nome_da_coluna_seq;
CREATE TABLE nome_da_tabela (
    nome_da_coluna integer DEFAULT nextval('nome_da_tabela_nome_da_coluna_seq') NOT NULL
);
```

Conforme visto, foi criada uma coluna do tipo inteiro e feito o valor padrão ser atribuído a partir de um gerador de sequência. A restrição `NOT NULL` é aplicada para garantir que não pode ser inserido o valor nulo explicitamente. Na maior parte das vezes, deve ser colocada uma restrição `UNIQUE` ou `PRIMARY KEY` para não permitir a inserção de valores duplicados por acidente, mas isto não é automático.

**Nota:** Antes do PostgreSQL 7.3 `serial` implicava `UNIQUE`, mas isto não é mais automático. Se for desejado que a coluna `serial` esteja em uma restrição de unicidade ou de chave primária isto deve ser especificado, da mesma forma como em qualquer outro tipo de dado.

Para inserir o próximo valor da seqüência em uma coluna do tipo `serial` deve ser especificada a atribuição do valor padrão à coluna `serial`, o que pode ser feito omitindo a coluna na lista de colunas no comando `INSERT`, ou através da utilização da palavra chave `DEFAULT`.

Os nomes de tipo `serial` e `serial4` são equivalentes: ambos criam colunas do tipo `integer`. Os nomes de tipo `bigserial` e `serial8` funcionam da mesma maneira, exceto por criarem uma coluna `bigint`. Deve ser utilizado `bigserial` se forem esperados mais de  $2^{31}$  identificadores durante a existência da tabela.

A seqüência criada para a coluna do tipo `serial` é removida automaticamente quando a coluna que a definiu é removida, e não pode ser removida de outra forma (Isto não era verdade nas versões do PostgreSQL anteriores a 7.3. Deve ser observado que este vínculo de remoção automática não ocorre em uma seqüência criada pela restauração da cópia de segurança de um banco de dados pré-7.3; a cópia de segurança não contém as informações necessárias para estabelecer o vínculo de dependência). Além disso, a dependência entre a seqüência e a coluna é feita apenas para a própria coluna `serial`; se qualquer outra coluna fizer referência à seqüência (talvez chamando manualmente a função `nextval()`), haverá rompimento se a seqüência for removida. Esta forma de utilizar as seqüências das colunas `serial` é considerada um estilo ruim. Se for desejado suprir várias colunas a partir do mesmo gerador de seqüência, a seqüência deve ser criada como um objeto independente.

### Exemplo 8-2. Alteração da seqüência da coluna `serial`

A seqüência criada para a coluna do tipo `serial` pode ter seus parâmetros alterados através do comando `ALTER SEQUENCE`, da mesma forma que qualquer outra seqüência criada através do comando `CREATE SEQUENCE`. Este exemplo mostra como proceder para fazer com que o valor inicial da coluna do tipo `serial` seja igual a 1000.<sup>7</sup>

```
=> CREATE TABLE t ( c1 SERIAL, c2 TEXT);
```

```
NOTICE: CREATE TABLE will create implicit sequence "t_c1_seq" for "serial" column "t.c1"
CREATE TABLE
```

```
=> \ds
```

```

                Lista de relações
Esquema |  Nome  |  Tipo   |  Dono
-----+-----+-----+-----
public  | t_c1_seq | seqüência | postgres
(1 linha)
```

```
=> ALTER SEQUENCE t_c1_seq RESTART WITH 1000;
```

```
=> INSERT INTO t VALUES (DEFAULT, 'Primeira linha');
```

```
=> SELECT * FROM t;
```

```

 c1 |      c2
----+-----
1000 | Primeira linha
(1 linha)
```

## 8.2. Tipos monetários

**Nota:** O tipo `money` está em obsolescência. Em seu lugar deve ser utilizado o tipo `numeric` ou `decimal`, em combinação com a função `to_char`.

O tipo `money` armazena a quantia monetária com uma precisão fracionária fixa; consulte a Tabela 8-3. A entrada é aceita em vários formatos, incluindo literais inteiros e de ponto flutuante, e também o formato monetário “típico”, como ‘\$1,000.00’. A saída geralmente é neste último formato, mas depende do idioma).

Tabela 8-3. Tipos monetários

Nome	Tamanho de Armazenamento	Descrição	Faixa
money	4 bytes	quantia monetária	-21474836.48 a +21474836.47

### 8.3. Tipos para cadeias de caracteres

A Tabela 8-4 mostra os tipos de propósito geral para cadeias de caracteres disponíveis no PostgreSQL.

Tabela 8-4. Tipos para cadeias de caracteres

Nome	Descrição
<code>character varying(n)</code> , <code>varchar(n)</code>	comprimento variável com limite
<code>character(n)</code> , <code>char(n)</code>	comprimento fixo, completado com brancos
<code>text</code>	comprimento variável não limitado

O SQL define dois tipos primários para caracteres: `character varying(n)` e `character(n)`, onde  $n$  é um número inteiro positivo. Estes dois tipos podem armazenar cadeias de caracteres com comprimento de até  $n$  caracteres. A tentativa de armazenar uma cadeia de caracteres mais longa em uma coluna de um destes tipos resulta em erro, a não ser que os caracteres excedentes sejam todos espaços; neste caso a cadeia de caracteres será truncada em seu comprimento máximo (Esta exceção um tanto bizarra é requerida pelo padrão SQL). Se a cadeia de caracteres a ser armazenada for mais curta que o comprimento declarado, os valores do tipo `character` são completados com espaços; os valores do tipo `character varying` simplesmente armazenam uma cadeia de caracteres mais curta.

Se um valor for convertido explicitamente (`cast`) para `character varying(n)`, ou para `character(n)`, o excesso de comprimento será truncado para  $n$  caracteres sem gerar erro (isto também é requerido pelo padrão SQL).

**Nota:** Antes do PostgreSQL 7.2 as cadeias de caracteres muito longas eram sempre truncadas sem gerar erro, tanto no contexto de conversão explícita quanto no de implícita.

As notações `varchar(n)` e `char(n)` são sinônimos para `character varying(n)` e `character(n)`, respectivamente. O uso de `character` sem especificação de comprimento equivale a `character(1)`; se for utilizado `character varying` sem especificador de comprimento, este tipo aceita cadeias de caracteres de qualquer tamanho. Este último é uma extensão do PostgreSQL.

Além desses o PostgreSQL disponibiliza o tipo `text`, que armazena cadeias de caracteres de qualquer comprimento. Embora o tipo `text` não esteja no padrão SQL, vários outros sistemas gerenciadores de banco de dados SQL também o possuem.

Os valores do tipo `character` são preenchidos fisicamente com espaços até o comprimento  $n$  especificado, sendo armazenados e mostrados desta forma. Entretanto, os espaços de preenchimento são tratados como não sendo significativos semanticamente. Os espaços de preenchimento são desconsiderados ao se comparar dois valores do tipo `character`, e são removidos ao converter um valor do tipo `character` para um dos outros tipos para cadeia de caracteres. Deve ser observado que os espaços no final *são* significativos semanticamente nos valores dos tipos `character varying` e `text`.

São necessários para armazenar dados destes tipos 4 bytes mais a própria cadeia de caracteres e, no caso do tipo `character`, mais os espaços para completar o tamanho. As cadeias de caracteres longas são comprimidas automaticamente pelo sistema e, portanto, o espaço físico necessário em disco pode ser menor. Os valores longos também são armazenados em tabelas secundárias, para não interferirem com o acesso rápido aos valores mais curtos da coluna. De qualquer forma, a cadeia de caracteres mais longa que pode ser armazenada é em torno de 1 GB (O valor máximo permitido para  $n$  na declaração do tipo de dado é menor que isto. Não seria muito útil mudar, porque de todo jeito nas codificações de caractere multibyte o número de caracteres e de bytes podem ser bem diferentes. Se for desejado armazenar cadeias de caracteres longas, sem um limite superior especificado, deve ser utilizado `text` ou `character varying` sem a especificação de comprimento, em vez de especificar um limite de comprimento arbitrário).

**Dica:** Não existe diferença de desempenho entre estes três tipos, a não ser pelo aumento do tamanho do armazenamento quando é utilizado o tipo completado com brancos. Enquanto o tipo `character(n)` possui vantagens

no desempenho em alguns outros sistemas gerenciadores de banco de dados, não possui estas vantagens no PostgreSQL. Na maioria das situações deve ser utilizado `text` ou `character varying` em vez deste tipo.

Consulte a Seção 4.1.2.1 para obter informações sobre a sintaxe dos literais cadeias de caracteres, e o Capítulo 9 para obter informações sobre os operadores e funções. O conjunto de caracteres do banco de dados determina o conjunto de caracteres utilizado para armazenar valores textuais; para obter mais informações sobre o suporte a conjunto de caracteres consulte a Seção 20.2.

Existem dois outros tipos para cadeias de caracteres de comprimento fixo no PostgreSQL, mostrados na Tabela 8-5. O tipo `name` existe *apenas* para armazenamento de identificadores nos catálogos internos do sistema, não tendo por finalidade ser usado pelos usuários comuns. Seu comprimento é definido atualmente como 64 bytes (63 caracteres utilizáveis mais o terminador) mas deve ser referenciado utilizando a constante `NAMEDATALEN`. O comprimento é definido quando é feita a compilação (sendo, portanto, ajustável para usos especiais); o padrão para comprimento máximo poderá mudar em uma versão futura. O tipo `"char"` (observe as aspas) é diferente de `char(1)`, porque utiliza apenas um byte para armazenamento. É utilizado internamente nos catálogos do sistema como o tipo de enumeração do homem pobre (`poor-man's enumeration type`).

**Tabela 8-5. Tipos especiais para caracteres**

Nome	Tamanho de Armazenamento	Descrição
"char"	1 byte	tipo interno de um único caractere
name	64 bytes	tipo interno para nomes de objeto

#### Exemplo 8-3. Utilização dos tipos para cadeias de caracteres

```
=> CREATE TABLE teste1 (a character(4));
=> INSERT INTO teste1 VALUES ('ok');
=> SELECT a, char_length(a) FROM teste1; -- ❶
```

```

a   | char_length
-----+-----
ok  |          4
```

```

=> CREATE TABLE teste2 (b VARCHAR(5));
=> INSERT INTO teste2 VALUES ('ok');
=> INSERT INTO teste2 VALUES ('bom      '); -- ❷
=> INSERT INTO teste2 VALUES ('muito longo');
ERRO:  valor muito longo para o tipo character varying(5)
=> INSERT INTO teste2 VALUES (CAST('muito longo' AS VARCHAR(5))); -- truncamento explícito
=> SELECT b, char_length(b) FROM teste2;
```

```

b   | char_length
-----+-----
ok  |          2
bom |          5
muito |          5
```

- ❶ A função `char_length` é mostrada na Seção 9.4.
- ❷ O DB2 8.1 atua da mesma maneira que o PostgreSQL 8.0.0, truncando os espaços à direita que excedem o tamanho do campo, o SQL Server 2000 também, mas a função `len` não conta os espaços à direita e o comprimento mostrado fica sendo igual a 3, enquanto o Oracle 10g não trunca os espaços à direita e gera mensagem de erro informando que o valor é muito longo, como no comando seguinte. (N. do T.)

#### Exemplo 8-4. Comparação de cadeias de caracteres com espaço à direita

Nestes exemplos faz-se a comparação de uma cadeia de caracteres com espaço à direita com outra cadeia de caracteres idêntica sem espaço à direita. Na tabela `t1` é feita a comparação entre dois tipos `char`, na tabela `t2` é feita a comparação

entre dois tipos varchar, e na tabela t3 é feita a comparação entre os tipos char e varchar. O mesmo script foi executado no PostgreSQL, no Oracle, no SQL Server e no DB2. Abaixo está mostrado o script executado: <sup>8 9</sup>

```
CREATE TABLE t1 ( c1 CHAR(10), c2 CHAR(10));
INSERT INTO t1 VALUES ('X', 'X ');
SELECT ''' || c1 || ''' AS c1,
       ''' || c2 || ''' AS c2,
       CASE WHEN (c1=c2) THEN 'igual' ELSE 'diferente' END AS comparação
FROM t1;

CREATE TABLE t2 ( c1 VARCHAR(10), c2 VARCHAR(10));
INSERT INTO t2 VALUES ('X', 'X ');
SELECT ''' || c1 || ''' AS c1,
       ''' || c2 || ''' AS c2,
       CASE WHEN (c1=c2) THEN 'igual' ELSE 'diferente' END AS comparação
FROM t2;

CREATE TABLE t3 ( c1 CHAR(10), c2 VARCHAR(10));
INSERT INTO t3 VALUES ('X', 'X ');
INSERT INTO t3 VALUES ('X ', 'X');
SELECT ''' || c1 || ''' AS c1,
       ''' || c2 || ''' AS c2,
       CASE WHEN (c1=c2) THEN 'igual' ELSE 'diferente' END AS comparação
FROM t3;
```

A seguir estão mostrados os resultados obtidos:

PostgreSQL 8.0.0:

c1	c2	comparação
'X'	'X'	igual

c1	c2	comparação
'X'	'X '	diferente

c1	c2	comparação
'X'	'X '	igual
'X'	'X'	igual

SQL Server 2000:

c1	c2	comparação
'X'	'X'	igual

c1	c2	comparação
'X'	'X '	igual

c1	c2	comparação
'X'	'X '	igual
'X'	'X'	igual

Oracle 10g:

C1	C2	COMPARAÇÃO
'X'	'X'	igual

C1	C2	COMPARAÇÃO
-----	-----	-----
'X'	'X '	diferente

C1	C2	COMPARAÇÃO
-----	-----	-----
'X	' 'X '	diferente
'X	' 'X'	diferente

DB2 8.1:

C1	C2	COMPARAÇÃO
-----	-----	-----
'X	' 'X	' igual

C1	C2	COMPARAÇÃO
-----	-----	-----
'X'	'X '	igual

C1	C2	COMPARAÇÃO
-----	-----	-----
'X	' 'X '	igual
'X	' 'X'	igual

Como pode ser visto, no SQL Server e no DB2 todas as comparações foram consideradas como sendo iguais. No Oracle só foi considerada igual a comparação entre dois tipos `char`, enquanto no PostgreSQL só foi considerada diferente a comparação entre dois tipos `varchar`.

## 8.4. Tipos de dado binários

O tipo de dado `bytea` permite o armazenamento de cadeias binárias; consulte a Tabela 8-6.

**Tabela 8-6. Tipos de dado binários**

Nome	Tamanho de Armazenamento	Descrição
bytea	4 bytes mais a cadeia binária	Cadeia binária de comprimento variável

A cadeia binária é uma sequência de octetos (ou bytes). As cadeias binárias se distinguem das cadeias de caracteres por duas características: Em primeiro lugar, as cadeias binárias permitem especificamente o armazenamento de octetos com o valor zero e outros octetos “não-imprimíveis” (geralmente octetos fora da faixa 32 a 126). As cadeias de caracteres não permitem octetos zero, e também não permitem outros valores de octeto e seqüências de valores de octeto inválidas de acordo com o conjunto de caracteres da codificação selecionada para o banco de dados. Em segundo lugar, as operações nas cadeias binárias processam os bytes como estão armazenados, enquanto o processamento das cadeias de caracteres dependem do idioma definido. Resumindo, as cadeias binárias são apropriadas para armazenar dados que os programadores imaginam como “octetos crus” (*raw bytes*), enquanto as cadeias de caracteres são apropriadas para armazenar texto.

Ao entrar com valores para `bytea` octetos com certos valores *devem* estar numa seqüência de escape (porém, todos os valores de octeto *podem* estar numa seqüência de escape) quando utilizados como parte do literal cadeia de bytes em uma declaração SQL. Em geral, para construir a seqüência de escape de um octeto este é convertido em um número octal de três dígitos equivalente ao valor decimal do octeto, e precedido por duas contrabarras. A Tabela 8-7 mostra os caracteres que devem estar em uma seqüência de escape, e fornece a seqüência de escape alternativa onde aplicável.

**Tabela 8-7. Octetos com seqüência de escape para literais `bytea`**

Valor decimal do octeto	Descrição	Representação da entrada com escape	Exemplo	Representação da saída
0	octeto zero	'\\000'	SELECT '\\000'::bytea;	\000
39	apóstrofo	'\'' ou '\\047'	SELECT	'

Valor decimal do octeto	Descrição	Representação da entrada com escape	Exemplo	Representação da saída
			'\'::bytea;	
92	contrabarra	'\\\' ou '\\134'	SELECT '\\\'::bytea;	\\
0 a 31 e 127 a 255	octetos “não-imprimíveis”	'\\xxx' (valor octal)	SELECT '\\001'::bytea;	\\001

A necessidade de colocar os octetos “não-imprimíveis” em uma sequência de escape varia conforme o idioma definido. Em certas circunstâncias podem ser deixados fora de uma sequência de escape. Deve ser observado que o resultado de todos os exemplos da Tabela 8-7 têm exatamente um octeto de comprimento, muito embora a representação de saída do octeto zero e da contrabarra possuam mais de um caractere.

#### Exemplo 8-5. Letra acentuada em bytea

Neste exemplo é feita a conversão explícita da cadeia de caracteres `aaaa` para o tipo `bytea`. Os mesmos resultados são obtidos em bancos de dados com conjunto de caracteres `LATIN1` e `SQL_ASCII`, desde que o conjunto de caracteres do cliente seja `ISO-8859-1` ou `1252` (Windows). A letra `a` sem acento é mostrada literalmente, mas as letras acentuadas são mostradas através de valores octais.<sup>10 11</sup>

```
=> \!chcp 1252
Active code page: 1252
=> SELECT cast('aaaa' AS bytea);
```

```

      bytea
-----
a\340\341\342\343
(1 linha)
```

O motivo pelo qual é necessário escrever tantas contrabarras, conforme mostrado na Tabela 8-7, é que uma cadeia de caracteres de entrada escrita como um literal cadeia de caracteres deve passar por duas fases de análise no servidor PostgreSQL. A primeira contrabarra de cada par é interpretada como um caractere de escape pelo analisador de literais cadeias de caracteres e portanto consumida, deixando a segunda contrabarra do par. A contrabarra remanescente é então reconhecida pela função de entrada de `bytea` como o início de um valor octal de três dígitos ou como escape de outra contrabarra. Por exemplo, o literal cadeia de caracteres passado para o servidor como `'\\001'` se torna `'\001'` após passar pelo analisador de literais cadeias de caracteres. O `'\001'` é então enviado para a função de entrada de `bytea`, onde é convertido em um único octeto com valor decimal igual a 1. Deve ser observado que o caractere apóstrofo não recebe tratamento especial por `bytea` e, portanto, segue as regras usuais para literais cadeias de caracteres (Consulte também a Seção 4.1.2.1.)

Os octetos `bytea` também são transformados em sequências de escape na saída. De uma maneira geral, cada octeto “não-imprimível” é convertido em seu valor octal equivalente de três dígitos, e precedido por uma contrabarra. Os octetos “imprimíveis” são, em sua maioria, representados através de sua representação padrão no conjunto de caracteres do cliente. O octeto com valor decimal 92 (contrabarra) possui uma representação de saída alternativa especial. Os detalhes podem ser vistos na Tabela 8-8.

**Tabela 8-8. Saída dos octetos `bytea` com escape**

Valor decimal do octeto	Descrição	Representação da saída com escape	Exemplo	Resultado de saída
92	contrabarra	\\	SELECT '\\134'::bytea;	\\
0 a 31 e 127 a 255	octetos “não-imprimíveis”	\\xxx (valor octal)	SELECT '\\001'::bytea;	\\001



Valor decimal do octeto	Descrição	Representação da saída com escape	Exemplo	Resultado de saída
32 a 126	octetos “imprimíveis”	representação no conjunto de caracteres do cliente	<code>SELECT '\\176':::bytea;</code>	~

Dependendo do programa cliente do PostgreSQL utilizado, pode haver trabalho adicional a ser realizado em relação a colocar e retirar escapes das cadeias `bytea`. Por exemplo, pode ser necessário colocar escapes para os caracteres de nova-linha e retorno-de-carro se a interface realizar a tradução automática destes caracteres.

O padrão SQL define um tipo de cadeia binária diferente, chamado `BLOB` ou `BINARY LARGE OBJECT` (objeto binário grande). O formato de entrada é diferente se comparado com `bytea`, mas as funções e operadores fornecidos são praticamente os mesmos.

## 8.5. Tipos para data e hora

O PostgreSQL suporta o conjunto completo de tipos para data e hora do SQL, mostrados na Tabela 8-9. As operações disponíveis para estes tipos de dado estão descritas na Seção 9.9.

**Tabela 8-9. Tipos para data e hora**

Nome	Tamanho de Armazenamento	Descrição	Menor valor	Maior valor	Resolução
<code>timestamp [ (p) ] [ without time zone ]</code>	8 bytes	tanto data quanto hora	4713 AC	5874897 DC	1 microssegundo / 14 dígitos
<code>timestamp [ (p) ] with time zone</code>	8 bytes	tanto data quanto hora, com zona horária	4713 AC	5874897 DC	1 microssegundo / 14 dígitos
<code>interval [ (p) ]</code>	12 bytes	intervalo de tempo	-178000000 anos	178000000 anos	1 microssegundo / 14 dígitos
<code>date</code>	4 bytes	somente data	4713 AC	32767 DC	1 dia
<code>time [ (p) ] [ without time zone ]</code>	8 bytes	somente a hora do dia	00:00:00.00	23:59:59.99	1 microssegundo / 14 dígitos
<code>time [ (p) ] with time zone</code>	12 bytes	somente a hora do dia, com zona horária	00:00:00.00+12	23:59:59.99-12	1 microssegundo / 14 dígitos

**Nota:** Antes do PostgreSQL 7.3, escrever apenas `timestamp` equivalia a escrever `timestamp with time zone`. Isto foi mudado para ficar em conformidade com o padrão SQL.

Os tipos `time`, `timestamp`, e `interval` aceitam um valor opcional de precisão *p*, que especifica o número de dígitos fracionários mantidos no campo de segundos. Por padrão não existe limite explícito para a precisão. O intervalo permitido para *p* é de 0 a 6 para os tipos `timestamp` e `interval`.

**Nota:** Quando os valores de `timestamp` são armazenados como números de ponto flutuante de precisão dupla (atualmente o padrão), o limite efetivo da precisão pode ser inferior a 6. Os valores de `timestamp` são armazenados como segundos antes ou após a meia-noite de 2000-01-01. A precisão de microssegundos é obtida para datas próximas a 2000-01-01 (alguns anos), mas a precisão degrada para datas mais afastadas. Quando os valores de `timestamp` são armazenadas como inteiros de oito bytes (uma opção de compilação), a precisão de microssegundo está disponível para toda a faixa de valores. Entretanto, os valores de `timestamp` em inteiros de 8 bytes possuem uma faixa de tempo mais limitada do que a mostrada acima: de 4713 AC até 294276 DC. A mesma opção de compilação também determina se os valores de `time` e `interval` são armazenados como ponto flutuante ou inteiros de oito bytes. No caso de ponto flutuante, a precisão dos valores de `interval` degrada conforme o tamanho do intervalo aumenta.

Para os tipos `time` o intervalo permitido para  $p$  é de 0 a 6 quando armazenados em inteiros de oito bytes, e de 0 a 10 quando armazenados em ponto flutuante.

O tipo `time with time zone` é definido pelo padrão SQL, mas a definição contém propriedades que levam a uma utilidade duvidosa. Na maioria dos casos, a combinação de `date`, `time`, `timestamp without time zone` e `timestamp with time zone` deve fornecer uma faixa completa de funcionalidades para data e hora requeridas por qualquer aplicativo.

Os tipos `abstime` e `reltime` são tipos de menor precisão usados internamente. É desestimulada a utilização destes tipos em novos aplicativos, além de ser incentivada a migração dos aplicativos antigos quando apropriado. Qualquer um destes tipos internos pode desaparecer em uma versão futura, ou mesmo todos.

### 8.5.1. Entrada de data e hora

A entrada da data e da hora é aceita em praticamente todos os formatos razoáveis, incluindo o ISO 8601, o SQL-compatível, o POSTGRES tradicional, além de outros. Para alguns formatos a ordem do dia, mês e ano na entrada da data é ambíguo e, por isso, existe suporte para especificar a ordem esperada destes campos. Deve ser definido o parâmetro `DateStyle` como `MDY` para selecionar a interpretação mês-dia-ano, `DMY` para selecionar a interpretação dia-mês-ano, ou `YMD` para selecionar a interpretação ano-mês-dia.

O PostgreSQL é mais flexível no tratamento da entrada de data e hora do que o requerido pelo padrão SQL. Consulte o Apêndice B para conhecer as regras exatas de análise da entrada de data e hora e os campos texto reconhecidos, incluindo meses, dias da semana e zonas horárias.

Lembre-se que qualquer entrada literal de data ou hora necessita estar entre apóstrofes, como os textos das cadeias de caracteres. Consulte a Seção 4.1.2.5 para obter informações adicionais. O SQL requer a seguinte sintaxe

```
tipo [ (p) ] 'valor'
```

onde  $p$ , na especificação opcional da precisão, é um número inteiro correspondendo ao número de dígitos fracionários do campo de segundos. A precisão pode ser especificada para os tipos `time`, `timestamp` e `interval`. Os valores permitidos estão mencionados acima. Se não for especificada nenhuma precisão na especificação da constante, a precisão do valor literal torna-se o padrão.

#### 8.5.1.1. Datas

A Tabela 8-10 mostra algumas entradas possíveis para o tipo `date`.

**Tabela 8-10. Entrada de data**

Exemplo	Descrição
January 8, 1999	não-ambíguo em qualquer modo de entrada em <code>datestyle</code>
1999-01-08	ISO 8601; 8 de janeiro em qualquer modo (formato recomendado)
1/8/1999	8 de janeiro no modo <code>MDY</code> ; 1 de agosto no modo <code>DMY</code>
1/18/1999	18 de janeiro no modo <code>MDY</code> ; rejeitado nos demais modos
01/02/03	2 de janeiro de 2003 no modo <code>MDY</code> ; 1 de fevereiro de 2003 no modo <code>DMY</code> ; 3 de fevereiro de 2001 no modo <code>YMD</code>
1999-Jan-08	8 de janeiro e qualquer modo
Jan-08-1999	January 8 em qualquer modo
08-Jan-1999	8 de janeiro em qualquer modo
99-Jan-08	8 de janeiro no modo <code>YMD</code> , caso contrário errado
08-Jan-99	8 de janeiro, porém errado no modo <code>YMD</code>
Jan-08-99	8 de janeiro, porém errado no modo <code>YMD</code>
19990108	ISO 8601; 8 de janeiro de 1999 em qualquer modo

Exemplo	Descrição
990108	ISO 8601; 8 de janeiro de 1999 em qualquer modo
1999.008	ano e dia do ano
J2451187	dia juliano
January 8, 99 BC	ano 99 antes da era comum <sup>a</sup>
Notas: a. A Era Comum (EC), ou “Common Era (CE)” em inglês, é um termo relativamente novo que tem experimentado um aumento de utilização e se espera que, eventualmente, substitua AD. Este último é uma abreviação de “Anno Domini” em Latim, ou “Ano do Senhor”. Este último se refere ao ano de nascimento aproximado de Yeshua de Nazaré (Jesus Cristo). EC, CE, AD e DC possuem o mesmo valor. The use of "CE" and "BCE" to identify dates ( <a href="http://www.religioustolerance.org/ce.htm">http://www.religioustolerance.org/ce.htm</a> ) (N. do T.)	

### 8.5.1.2. Horas

Os tipos hora-do-dia são `time [ (p) ] without time zone` e `time [ (p) ] with time zone`. Escrever apenas `time` equivale a escrever `time without time zone`.

Entradas válidas para estes tipos consistem na hora do dia seguida por uma zona horária opcional (Consulte a Tabela 8-11 e a Tabela 8-12). Se for especificada a zona horária na entrada de `time without time zone`, esta é ignorada em silêncio.

**Tabela 8-11. Entrada de hora**

Exemplo	Descrição
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	o mesmo que 04:05; AM não afeta o valor
04:05 PM	o mesmo que 16:05; a hora entrada deve ser ≤ 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	zona horária especificada pelo nome

**Tabela 8-12. Entrada de zona horária**

Exemplo	Descrição
PST	Hora Padrão do Pacífico (Pacific Standard Time)
-8:00	deslocamento ISO-8601 para PST
-800	deslocamento ISO-8601 para PST
-8	deslocamento ISO-8601 para PST
zulu	Abreviatura militar para UTC
z	Forma abreviada de zulu

Consulte o Apêndice B para ver a lista de nomes de zona horária reconhecidos na entrada.

### 8.5.1.3. Carimbos do tempo

As entradas válidas para os tipos carimbo do tempo são formadas pela concatenação da data com a hora seguida, opcionalmente, pela zona horária, e seguida opcionalmente por AD ou BC (Como alternativa, AD ou BC pode aparecer antes da zona horária, mas esta não é a ordem preferida). Portanto,

```
1999-01-08 04:05:06
```

e

```
1999-01-08 04:05:06 -8:00
```

são valores válidos, que seguem o padrão ISO 8601. Além desses, é suportado o formato muito utilizado

```
January 8 04:05:06 1999 PST
```

O padrão SQL diferencia os literais `timestamp without time zone` de `timestamp with time zone` pela existência de “+”; ou “-”. Portanto, de acordo com o padrão,

```
TIMESTAMP '2004-10-19 10:23:54'
```

é um `timestamp without time zone`, enquanto

```
TIMESTAMP '2004-10-19 10:23:54+02'
```

é um `timestamp with time zone`. O PostgreSQL difere do padrão requerendo que os literais `timestamp with time zone` sejam digitados explicitamente:

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

Se o literal não for informado explicitamente como sendo `timestamp with time zone`, o PostgreSQL ignora em silêncio qualquer indicação de zona horária no literal. Ou seja, o valor resultante de data e hora é derivado dos campos `data` e `hora` do valor da entrada, não sendo ajustado conforme a zona horária.

Para `timestamp with time zone`, o valor armazenado internamente está sempre em UTC (Tempo Universal Coordenado, tradicionalmente conhecido por Hora Média de Greenwich, GMT<sup>12</sup>). Um valor de entrada possuindo a zona horária especificada explicitamente é convertido em UTC utilizando o deslocamento apropriado para esta zona horária. Se não for especificada nenhuma zona horária na cadeia de caracteres da entrada, pressupõe-se que está na mesma zona horária indicada pelo parâmetro do sistema `timezone`, sendo convertida em UTC utilizando o deslocamento da zona em `timezone`.

Quando um valor de `timestamp with time zone` é enviado para a saída, é sempre convertido de UTC para a zona horária corrente de `timezone`, e mostrado como hora local desta zona. Para ver a hora em outra zona horária, ou se muda `timezone` ou se usa a construção `AT TIME ZONE` (consulte a Seção 9.9.3).

As conversões entre `timestamp without time zone` e `timestamp with time zone` normalmente assumem que os valores de `timestamp without time zone` devem ser recebidos ou fornecidos como hora local da `timezone`. A referência para uma zona horária diferente pode ser especificada para a conversão utilizando `AT TIME ZONE`.

### 8.5.1.4. Intervalos

Os valores do tipo `interval` podem ser escritos utilizando uma das seguintes sintaxes:

```
[@] quantidade unidade [quantidade unidade...] [direção]
```

onde: *quantidade* é um número (possivelmente com sinal); *unidade* é `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, `millennium`, ou abreviaturas ou plurais destas unidades; *direção* pode ser `ago` (atrás) ou vazio. O sinal de arroba (@) é opcional. As quantidades com unidades diferentes são implicitamente adicionadas na conta com o sinal adequado.

As quantidades de dias, horas, minutos e segundos podem ser especificadas sem informar explicitamente as unidades. Por exemplo, `'1 12:59:10'` é lido do mesmo modo que `'1 day 12 hours 59 min 10 sec'`.

A precisão opcional *p* deve estar entre 0 e 6, sendo usado como padrão a precisão do literal da entrada.

### 8.5.1.5. Valores especiais

Por ser conveniente, o PostgreSQL também suporta vários valores especiais para entrada de data e hora, conforme mostrado na Tabela 8-13. Os valores *infinity* e *-infinity* possuem representação especial dentro do sistema, sendo mostrados da mesma maneira; porém, os demais são simplesmente notações abreviadas convertidas para valores comuns de data e hora ao serem lidos (Em particular, *now* e as cadeias de caracteres relacionadas são convertidas para um valor específico de data e hora tão logo são lidas). Todos estes valores devem ser escritos entre apóstrofes quando usados como constantes nos comandos SQL.

**Tabela 8-13. Entradas especiais de data e hora**

Cadeia de caracteres entrada	Tipos válidos	Descrição
epoch	date, timestamp	1970-01-01 00:00:00+00 (hora zero do sistema Unix)
infinity	timestamp	mais tarde que todos os outros carimbos do tempo
-infinity	timestamp	mais cedo que todos os outros carimbos do tempo
now	date, time, timestamp	hora de início da transação corrente
today	date, timestamp	meia-noite de hoje
tomorrow	date, timestamp	meia-noite de amanhã
yesterday	date, timestamp	meia-noite de ontem
allballs	time	00:00:00.00 UTC

Também podem ser utilizadas as seguintes funções, compatíveis com o padrão SQL, para obter o valor corrente de data e hora para o tipo de dado correspondente: *CURRENT\_DATE*, *CURRENT\_TIME*, *CURRENT\_TIMESTAMP*, *LOCALTIME* e *LOCALTIMESTAMP*. As últimas quatro aceitam, opcionalmente, a especificação da precisão (consulte também a Seção 9.9.4). Entretanto, deve ser observado que são funções SQL, *não* sendo reconhecidas como cadeias de caracteres de entrada de dados.

### Exemplo 8-6. Utilização das entradas especiais de data e hora

Neste exemplo são mostradas utilizações das entradas especiais de data e hora para o tipo *timestamp with time zone*.

13

```
=> CREATE TABLE t ( c1 TEXT, c2 TIMESTAMP WITH TIME ZONE );
=> BEGIN;
=> INSERT INTO t VALUES ('epoch', 'epoch');
=> INSERT INTO t VALUES ('infinity', 'infinity');
=> INSERT INTO t VALUES ('-infinity', '-infinity');
=> INSERT INTO t VALUES ('now', 'now');
=> INSERT INTO t VALUES ('today', 'today');
=> INSERT INTO t VALUES ('tomorrow', 'tomorrow');
=> INSERT INTO t VALUES ('yesterday', 'yesterday');
=> INSERT INTO t VALUES ('CURRENT_TIMESTAMP', CURRENT_TIMESTAMP);
=> END;
=> SELECT * FROM t;
```

c1	c2
epoch	1969-12-31 21:00:00-03
infinity	infinity
-infinity	-infinity
now	2005-04-19 18:20:35.164293-03
today	2005-04-19 00:00:00-03
tomorrow	2005-04-20 00:00:00-03
yesterday	2005-04-18 00:00:00-03

CURRENT\_TIMESTAMP | 2005-04-19 18:20:35.164293-03  
(8 linhas)

### 8.5.2. Saídas de data e hora

Utilizando o comando `SET datestyle` o formato de saída para os tipos de data e hora pode ser definido como um dos quatro estilos ISO 8601, SQL (Ingres), POSTGRES tradicional e German. O padrão é o formato ISO (o padrão SQL requer a utilização do formato ISO 8601; o nome do formato de saída “SQL” é um acidente histórico). A Tabela 8-14 mostra exemplo de cada um dos estilos de saída. A saída dos tipos `date` e `time` obviamente utilizam apenas a parte da data ou da hora de acordo com os exemplos fornecidos.

**Tabela 8-14. Estilos de saída de data e hora**

Especificação de estilo	Descrição	Exemplo
ISO	ISO 8601/padrão SQL	2005-04-21 18:39:28.283566-03
SQL	estilo tradicional	04/21/2005 18:39:28.283566 BRT
POSTGRES	estilo original	Thu Apr 21 18:39:28.283566 2005 BRT
German	estilo regional	21.04.2005 18:39:28.283566 BRT

Nos estilos SQL e POSTGRES, o dia vem antes do mês se a ordem de campo DMY tiver sido especificada, senão o mês vem antes do dia (veja na Seção 8.5.1 como esta especificação também afeta a interpretação dos valores de entrada). A Tabela 8-15 mostra um exemplo.

**Tabela 8-15. Convenções de ordem na data**

Definição de <code>datestyle</code>	Ordem de entrada	Exemplo de saída
SQL, DMY	<i>dia/mês/ano</i>	21/04/2005 18:39:28.283566 BRT
SQL, MDY	<i>mês/dia/ano</i>	04/21/2005 18:39:28.283566 BRT
Postgres, DMY	<i>dia/mês/ano</i>	Thu 21 Apr 18:39:28.283566 2005 BRT

A saída do tipo `interval` se parece com o formato da entrada, exceto que as unidades como `century` e `week` são convertidas em anos e dias, e que `ago` é convertido no sinal apropriado. No modo ISO a saída se parece com

```
[ quantidade unidade [ ... ] ] [ dias ] [ horas:minutos:segundos ]
```

Os estilos de data e hora podem ser selecionados pelo usuário utilizando o comando `SET datestyle`, o parâmetro `DateStyle` no arquivo de configuração `postgresql.conf`, ou a variável de ambiente `PGDATESTYLE` no servidor ou no cliente. A função de formatação `to_char` (consulte a Seção 9.8) também pode ser utilizada como uma forma mais flexível de formatar a saída de data e hora.

### 8.5.3. Zonas horárias

Zonas horárias e convenções de zonas horárias são influenciadas por decisões políticas, e não apenas pela geometria da Terra. As zonas horárias em torno do mundo se tornaram um tanto padronizadas durante o século XX, mas continuam propensas a mudanças arbitrárias, particularmente com relação a horários de inverno e de verão. Atualmente o PostgreSQL suporta as regras de horário de inverno e verão (`daylight-savings rules`) no período de tempo que vai de 1902 até 2038 (correspondendo à faixa completa de tempo do sistema Unix convencional). Datas fora desta faixa são consideradas como estando na “hora padrão” da zona horária selecionada, sem importar em que parte do ano se encontram.

O PostgreSQL se esforça para ser compatível com as definições do padrão SQL para o uso típico. Entretanto, o padrão SQL possui uma combinação única de tipos e funcionalidades para data e hora. Os dois problemas óbvios são:

- Embora o tipo `date` não possua zona horária associada, o tipo `time` pode possuir. As zonas horárias do mundo real possuem pouco significado a menos que estejam associadas a uma data e hora, porque o deslocamento pode variar durante o ano devido ao horário de verão.

- A zona horária padrão é especificada como um deslocamento numérico constante em relação ao UTC. Não é possível, portanto, fazer ajuste devido ao horário de verão (DST) ao se realizar aritmética de data e hora entre fronteiras do horário de verão (DST).<sup>14</sup>

Para superar estas dificuldades, recomenda-se utilizar tipos de data e hora contendo tanto a data quanto a hora quando utilizar zonas horárias. Recomenda-se *não* utilizar o tipo `time with time zone` (embora seja suportado pelo PostgreSQL para os aplicativos legados e para conformidade com o padrão SQL). O PostgreSQL assume a zona horária local para qualquer tipo contendo apenas a data ou a hora.

Todas as datas e horas com zona horária são armazenadas internamente em UTC. São convertidas para a hora local na zona especificada pelo parâmetro de configuração `timezone` antes de serem mostradas ao cliente.

O parâmetro de configuração `timezone` pode ser definido no arquivo `postgresql.conf`, ou por qualquer outro meio padrão descrito na Seção 16.4. Existem, também, várias outras formas especiais de defini-lo:

- Se `timezone` não for especificada no arquivo de configuração `postgresql.conf`, nem como uma chave da linha de comando do `postmaster`, o servidor tenta utilizar o valor da variável de ambiente `TZ` como a zona horária padrão. Se `TZ` não estiver definida, ou não contiver nenhum nome de zona horária conhecido pelo PostgreSQL, o servidor tenta determinar a zona horária padrão do sistema operacional verificando o comportamento da função `localtime()` da biblioteca C. A zona horária padrão é selecionada através da correspondência mais próximas entre as zonas horárias conhecidas pelo PostgreSQL.
- O comando SQL `SET TIME ZONE` define a zona horária para a sessão. Esta é uma forma alternativa de `SET TIMEZONE TO` com uma sintaxe mais compatível com a especificação SQL.
- Se a variável de ambiente `PGTZ` estiver definida no cliente, é utilizada pelos aplicativos `libpq` para enviar o comando `SET TIME ZONE` para o servidor durante a conexão.

Consulte o Apêndice B para obter a lista de zonas horárias disponíveis.

#### 8.5.4. Internamente

O PostgreSQL utiliza datas Julianas<sup>15</sup> para todos os cálculos de data e hora, porque possuem a boa propriedade de prever/calcular corretamente qualquer data mais recente que 4713 AC até bem distante no futuro, partindo da premissa que o ano possui 365,2425 dias.

As convenções de data anteriores ao século 19 são uma leitura interessante, mas não são suficientemente consistentes para permitir a codificação em rotinas tratadoras de data e hora.

### 8.6. Tipo booleano

O PostgreSQL disponibiliza o tipo SQL padrão `boolean`. O tipo `boolean` pode possuir apenas um dos dois estados: “verdade” ou “falso”. O terceiro estado, “desconhecido”, é representado pelo valor nulo do SQL.<sup>16 17</sup>

Os valores literais válidos para o estado “verdade” são:

```
TRUE
't'
'true'
'y'
'yes'
'1'
```

Para o estado “falso” podem ser utilizados os seguintes valores:

```
FALSE
'f'
'false'
'n'
'no'
'0'
```

A utilização das palavras chave `TRUE` e `FALSE` é preferida (e em conformidade com o padrão SQL).

**Exemplo 8-7. Utilização do tipo boolean**

```
CREATE TABLE testel (a boolean, b text);
INSERT INTO testel VALUES (TRUE, 'sic est');
INSERT INTO testel VALUES (FALSE, 'non est');
SELECT * FROM testel;
```

```
a |      b
---+-----
t | sic est
f | non est
```

```
SELECT * FROM testel WHERE a;
```

```
a |      b
---+-----
t | sic est
```

O Exemplo 8-7 mostra que os valores do tipo boolean são exibidos utilizando as letras t e f.

**Dica:** Os valores do tipo boolean não podem ser convertidos diretamente em outros tipos (por exemplo, `CAST (valor_booleano AS integer)` não funciona). A conversão pode ser feita utilizando a expressão `CASE: CASE WHEN valor_booleano THEN 'valor se for verdade' ELSE 'valor se for falso' END`. Consulte a Seção 9.13.

O tipo boolean utiliza 1 byte para seu armazenamento.

**Exemplo 8-8. Classificação do tipo boolean**

Segundo o padrão SQL o valor verdade é maior que o valor falso. O PostgreSQL considera o valor nulo maior que estes dois, conforme mostrado neste exemplo.<sup>18</sup>

```
=> \pset null -
=> CREATE TABLE t (b BOOLEAN);
=> INSERT INTO t VALUES(true);
=> INSERT INTO t VALUES(false);
=> INSERT INTO t VALUES(null);
=> SELECT * FROM t ORDER BY b;
```

```
b
---
f
t
-
(3 linhas)
```

## 8.7. Tipos geométricos

Os tipos de dado geométricos representam objetos espaciais bidimensionais. A Tabela 8-16 mostra os tipos geométricos disponíveis no PostgreSQL. O tipo mais fundamental, o ponto, forma a base para todos os outros tipos.

**Tabela 8-16. Tipos geométricos**

Nome	Tamanho de Armazenamento	Descrição	Representação
point	16 bytes	Ponto no plano	(x,y)
line	32 bytes	Linha infinita (não totalmente implementado)	((x1,y1),(x2,y2))
lseg	32 bytes	Segmento de linha finito	((x1,y1),(x2,y2))
box	32 bytes	Caixa retangular	((x1,y1),(x2,y2))
path	16+16n bytes	Caminho fechado (semelhante ao polígono)	((x1,y1),...)



Nome	Tamanho de Armazenamento	Descrição	Representação
<code>path</code>	16+16n bytes	Caminho aberto	$[(x1,y1),...]$
<code>polygon</code>	40+16n bytes	Polígono (semelhante ao caminho fechado)	$((x1,y1),...)$
<code>circle</code>	24 bytes	Círculo	$\langle (x,y),r \rangle$ (centro e raio)

Está disponível um amplo conjunto de funções e operadores para realizar várias operações geométricas, como escala, translação, rotação e determinar interseções, conforme explicadas na Seção 9.10.

### 8.7.1. Pontos

Os pontos são os blocos de construção bidimensionais fundamentais para os tipos geométricos. Os valores do tipo `point` são especificados utilizando a seguinte sintaxe:

```
( x , y )
  x , y
```

onde  $x$  e  $y$  são as respectivas coordenadas na forma de números de ponto flutuante.

### 8.7.2. Segmentos de linha

Os segmentos de linha (`lseg`) são representados por pares de pontos. Os valores do tipo `lseg` são especificado utilizando a seguinte sintaxe:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1      , x2 , y2
```

onde  $(x1,y1)$  e  $(x2,y2)$  são os pontos das extremidades do segmento de linha.

### 8.7.3. Caixas

As caixas são representadas por pares de pontos de vértices opostos da caixa. Os valores do tipo `box` são especificados utilizando a seguinte sintaxe:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1      , x2 , y2
```

onde  $(x1,y1)$  e  $(x2,y2)$  são quaisquer vértices opostos da caixa.

As caixas são mostradas utilizando a primeira sintaxe. Os vértices são reordenados na entrada para armazenar o vértice direito superior e, depois, o vértice esquerdo inferior. Podem ser especificados outros vértices da caixa, mas os vértices esquerdo inferior e direito superior são determinados a partir da entrada e armazenados.

### 8.7.4. Caminhos

Os caminhos são representados por listas de pontos conectados. Os caminhos podem ser *abertos*, onde o primeiro e o último ponto da lista não são considerados conectados, e *fechados*, onde o primeiro e o último ponto são considerados conectados.

Os valores do tipo `path` são especificados utilizando a seguinte sintaxe:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1      , ... , xn , yn )
    x1 , y1      , ... , xn , yn
```

onde os pontos são os pontos das extremidades dos segmentos de linha que compõem o caminho. Os colchetes (`[]`) indicam um caminho aberto, enquanto os parênteses (`()`) indicam um caminho fechado.

Os caminhos são mostrados utilizando a primeira sintaxe.

### 8.7.5. Polígonos

Os polígonos são representados por uma lista de pontos (os vértices do polígono). Provavelmente os polígonos deveriam ser considerados equivalentes aos caminhos fechados, mas são armazenados de forma diferente e possuem um conjunto próprio de rotinas de suporte.

Os valores do tipo `polygon` são especificados utilizando a seguinte sintaxe:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1 , ... , xn , yn )
x1 , y1 , ... , xn , yn
```

onde os pontos são os pontos das extremidades dos segmentos de linha compondo a fronteira do polígono.

Os polígonos são mostrados utilizando a primeira sintaxe.

### 8.7.6. Círculos

Os círculos são representados por um ponto central e um raio. Os valores do tipo `circle` são especificado utilizando a seguinte sintaxe:

```
< ( x , y ) , r >
( ( x , y ) , r )
( x , y ) , r
x , y , r
```

onde  $(x,y)$  é o centro e  $r$  é o raio do círculo.

Os círculos são mostrados utilizando a primeira sintaxe.

## 8.8. Tipos para endereço de rede

O PostgreSQL disponibiliza tipos de dado para armazenar endereços IPv4, IPv6 e MAC, conforme mostrado na Tabela 8-17. É preferível utilizar estes tipos em vez dos tipos de texto puro, porque estes tipos possuem verificação de erro na entrada, além de vários operadores e funções especializadas (consulte Seção 9.11).

**Tabela 8-17. Tipos para endereço de rede**

Nome	Tamanho de Armazenamento	Descrição
<code>cidr</code>	12 ou 24 bytes	redes IPv4 e IPv6
<code>inet</code>	12 ou 24 bytes	hospedeiros e redes IPv4 e IPv6
<code>macaddr</code>	6 bytes	endereço MAC

Ao ordenar os tipos de dado `inet` e `cidr`, os endereços IPv4 vêm sempre na frente dos endereços IPv6, inclusive os endereços IPv4 encapsulados ou mapeados em endereços IPv6, tais como `::10.2.3.4` ou `::ffff::10.4.3.2`.

#### 8.8.1. `inet`

O tipo de dado `inet` armazena um endereço de hospedeiro IPv4 ou IPv6 e, opcionalmente, a identificação da sub-rede onde se encontra, tudo em um único campo. A identificação da sub-rede é representada declarando quantos bits do endereço do hospedeiro representam o endereço de rede (a “máscara de rede”). Se a máscara de rede for 32 e o endereço for IPv4, então o valor não indica uma sub-rede, e sim um único hospedeiro. No IPv6 o comprimento do endereço é de 128 bits e, portanto, 128 bits especificam o endereço de um único hospedeiro. Deve ser observado que se for desejado aceitar apenas endereços de rede, deve ser utilizado o tipo `cidr` em vez do tipo `inet`.

O formato de entrada para este tipo é *endereço/y*, onde *endereço* é um endereço IPv4 ou IPv6, e *y* é o número de bits da máscara de rede. Se a parte */y* for deixada de fora, então a máscara de rede será 32 para IPv4 e 128 para IPv6, e o valor representa um único hospedeiro apenas. Ao ser mostrado, a porção */y* é suprimida se a máscara de rede especificar apenas um único hospedeiro.

### 8.8.2. cidr

O tipo *cidr* armazena uma especificação de rede IPv4 ou IPv6. Os formatos de entrada e de saída seguem as convenções do Classless Internet Domain Routing<sup>19</sup>. O formato para especificar redes é *endereço/y*, onde *endereço* é a rede representada por um endereço IPv4 ou IPv6, e *y* é o número de bits da máscara de rede. Se *y* for omitido, será calculado utilizando as premissas do sistema de numeração com classes antigo, exceto que será pelo menos suficientemente grande para incluir todos os octetos escritos na entrada. É errado especificar endereço de rede contendo bits definidos à direita da máscara de rede especificada.

A Tabela 8-18 mostra alguns exemplos.

**Tabela 8-18. Exemplos de entrada para o tipo *cidr***

Entrada <i>cidr</i>	Saída <i>cidr</i>	<code>abbrev(cidr)</code>
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

### 8.8.3. inet versus cidr

A diferença essencial entre os tipos de dado *inet* e *cidr* é que *inet* aceita valores com bits diferente de zero à direita da máscara de rede, enquanto *cidr* não aceita.

**Dica:** Caso não se goste do formato de saída para os valores de *inet* ou *cidr*, deve-se tentar utilizar as funções `host()`, `text()` e `abbrev()`.

### 8.8.4. macaddr

O tipo *macaddr* armazena endereços de MAC<sup>20</sup>, ou seja, endereços de hardware da placa Ethernet (embora os endereços de MAC sejam utilizados para outras finalidades também). A entrada é aceita em vários formatos habituais incluindo

```
'08002b:010203'
```

```
'08002b-010203'
```

```
'0800.2b01.0203'
'08-00-2b-01-02-03'
'08:00:2b:01:02:03'
```

sendo que todos especificam o mesmo endereço. Letras maiúsculas e minúsculas são aceitas para os dígitos de a a f. A saída é sempre na última forma mostrada.

Na distribuição do código fonte do PostgreSQL, o diretório `contrib/mac` contém ferramentas que podem ser utilizadas para fazer a correspondência entre endereços de MAC e nomes de fabricantes de hardware.

## 8.9. Tipos para cadeias de bits

As cadeias de bits são cadeias de zeros e uns. Podem ser usadas para armazenar ou visualizar máscaras de bits. Existem dois tipos de dado para bits no SQL: `bit(n)` e `bit varying(n)`, onde  $n$  é um número inteiro positivo.

Os dados para o tipo `bit` devem possuir exatamente o comprimento  $n$ ; é errado tentar armazenar cadeias de bits mais curtas ou mais longas. O tipo de dado `bit varying` possui um comprimento variável até o máximo de  $n$ ; cadeias mais longas são rejeitadas. Escrever `bit` sem o comprimento equivale a escrever `bit(1)`, enquanto `bit varying` sem a especificação do comprimento significa comprimento ilimitado.

**Nota:** Se for feita uma conversão explícita do valor de uma cadeia de bits para `bit(n)`, os bits serão truncados ou completados à direita com zeros para ficar exatamente com  $n$  bits, sem ocasionar erro. De forma semelhante, se for feita uma conversão explícita do valor de uma cadeia de bits para `bit varying(n)`, os bits serão truncados à direita se houver mais de  $n$  bits.

**Nota:** Antes do PostgreSQL 7.2, os dados do tipo `bit` eram sempre truncados em silêncio ou completados à direita com zeros, com ou sem uma conversão explícita. Este comportamento foi modificado para ficar em conformidade com o padrão SQL.

Consulte a Seção 4.1.2.3 para obter informações sobre a sintaxe das constantes do tipo cadeia de bits. Estão disponíveis operadores lógicos para `bit` e funções para manipulação de cadeias de bits; consulte o Capítulo 9.

### Exemplo 8-9. Utilização dos tipos para cadeia de bits

```
CREATE TABLE teste (a BIT(3), b BIT VARYING(5));
INSERT INTO teste VALUES (B'101', B'00');
INSERT INTO teste VALUES (B'10', B'101');
ERRO: comprimento da cadeia de bits 2 não corresponde ao tipo bit(3)
INSERT INTO teste VALUES (B'10'::bit(3), B'101');
SELECT * FROM teste;
```

a	b
101	00
100	101

## 8.10. Matrizes

O PostgreSQL permite que colunas de uma tabela sejam definidas como matrizes (arrays) multidimensionais de comprimento variável. Podem ser criadas matrizes de qualquer tipo de dado, nativo ou definido pelo usuário (Entretanto, ainda não são suportadas matrizes de tipos compostos ou domínios).

### 8.10.1. Declaração do tipo matriz

Para ilustrar a utilização do tipo matriz, é criada a tabela abaixo:

```
CREATE TABLE sal_emp (
    nome          text,
    pagamento_semanal integer[],
    agenda        text[][]
);
```

Conforme visto, o tipo de dado matriz é nomeado anexando colchetes (`[]`) ao nome do tipo de dado dos elementos da matriz. O comando acima cria uma tabela chamada `sal_emp`, contendo uma coluna do tipo `text` (`nome`), uma matriz unidimensional do tipo `integer` (`pagamento_semanal`), que representa o salário semanal do empregado, e uma matriz bidimensional do tipo `text` (`agenda`), que representa a agenda semanal do empregado.

A sintaxe de `CREATE TABLE` permite especificar o tamanho exato da matriz como, por exemplo:

```
CREATE TABLE jogo_da_velha (
    casa      integer[3][3]
);
```

Entretanto, a implementação atual não obriga que os limites de tamanho da matriz sejam respeitados — o comportamento é o mesmo das matrizes com comprimento não especificado.

Na verdade, a implementação atual também não obriga que o número de dimensões declarado seja respeitado. As matrizes de um determinado tipo de elemento são todas consideradas como sendo do mesmo tipo, não importando o tamanho ou o número de dimensões. Portanto, a declaração do número de dimensões ou tamanhos no comando `CREATE TABLE` é simplesmente uma documentação, que não afeta o comportamento em tempo de execução.

Pode ser utilizada uma sintaxe alternativa para matrizes unidimensionais, em conformidade com o padrão SQL:1999. A coluna `pagamento_semanal` pode ser definida como:

```
pagamento_semanal integer ARRAY[4],
```

Esta sintaxe requer uma constante inteira para designar o tamanho da matriz. Entretanto, como anteriormente, o PostgreSQL não obriga que os limites de tamanho da matriz sejam respeitados.

### 8.10.2. Entrada de valor matriz

Para escrever um valor matriz como uma constante literal, os valores dos elementos devem ser envoltos por chaves (`{}`) e separados por vírgulas (Quem conhece C pode ver que não é diferente da sintaxe da linguagem C para inicializar estruturas). Podem ser colocadas aspas (`"`) em torno de qualquer valor de elemento, sendo obrigatório caso o elemento contenha vírgulas ou chaves (abaixo são mostrados mais detalhes). Portanto, o formato geral de uma constante matriz é o seguinte:

```
'{ val1 delim val2 delim ... }'
```

onde *delim* é o caractere delimitador para o tipo, conforme registrado na sua entrada em `pg_type`. Entre todos os tipos de dado padrão fornecidos na distribuição do PostgreSQL, o tipo `box` usa o ponto-e-vírgula (`;`) mas todos os demais usam a vírgula (`,`). Cada *val* é uma constante do tipo do elemento da matriz, ou uma submatriz. Um exemplo de uma constante matriz é

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

Esta constante é uma matriz bidimensional, 3 por 3, formada por três submatrizes de inteiros.

(Estes tipos de constante matriz são, na verdade, apenas um caso especial do tipo genérico de constantes mostrado na Seção 4.1.2.5. A constante é inicialmente tratada como uma cadeia de caracteres e passada para a rotina de conversão de entrada de matriz. Pode ser necessária uma especificação explícita do tipo).

Agora podemos ver alguns comandos `INSERT`.

```
INSERT INTO sal_emp
VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"reunião", "almoço"}, {"reunião"}}');
ERRO:      matrizes multidimensionais devem ter expressões de matriz com dimensões correspondentes
```

Deve ser observado que as matrizes multidimensionais devem possuir tamanhos correspondentes para cada dimensão. Uma combinação errada causa uma mensagem de erro.

```
INSERT INTO sal_emp
VALUES ('Bill',
       '{10000, 10000, 10000, 10000}',
       '{{"reunião", "almoço"}, {"treinamento", "apresentação"}}');
```

```
INSERT INTO sal_emp
VALUES ('Carol',
       '{20000, 25000, 25000, 25000}',
       '{{"café da manhã", "consultoria"}, {"reunião", "almoço"}}');
```

Uma limitação da implementação atual de matriz, é que os elementos individuais da matriz não podem ser valores nulos SQL. Toda a matriz pode ser definida como nula, mas não pode existir uma matriz com alguns elementos nulos e outros não.

O resultado das duas inserções anteriores se parece com:

```
SELECT * FROM sal_emp;
```

nome	pagamento_semanal	agenda
Bill	{10000,10000,10000,10000}	{{reunião,almoço},{treinamento,apresentação}}
Carol	{20000,25000,25000,25000}	{{"café da manhã",consultoria},{reunião,almoço}}

(2 linhas)

Também pode ser utilizada a sintaxe de construtor de ARRAY:

```
INSERT INTO sal_emp
VALUES ('Bill',
       ARRAY[10000, 10000, 10000, 10000],
       ARRAY['reunião', 'almoço'], ['treinamento', 'apresentação']));
```

```
INSERT INTO sal_emp
VALUES ('Carol',
       ARRAY[20000, 25000, 25000, 25000],
       ARRAY['café da manhã', 'consultoria'], ['reunião', 'almoço']));
```

Deve ser observado que os elementos da matriz são constantes comuns ou expressões SQL; por exemplo, os literais cadeia de caracteres ficam entre apóstrofes, em vez de aspas como no caso de um literal matriz. A sintaxe do construtor de ARRAY é mostrada com mais detalhes na Seção 4.2.10.

### 8.10.3. Acesso às matrizes

Agora podemos efetuar algumas consultas na tabela. Primeiro, será mostrado como acessar um único elemento da matriz de cada vez. Esta consulta mostra os nomes dos empregados cujo pagamento mudou na segunda semana:

```
SELECT nome FROM sal_emp WHERE pagamento_semanal[1] <> pagamento_semanal[2];
```

nome
Carol

(1 linha)

Os números dos índices da matriz são escritos entre colchetes. Por padrão, o PostgreSQL utiliza a convenção de numeração baseada em um, ou seja, uma matriz de  $n$  elementos começa por `array[1]` e termina por `array[n]`.

Esta consulta mostra o pagamento da terceira semana de todos os empregados:

```
SELECT pagamento_semanal[3] FROM sal_emp;
```

pagamento_semanal
10000
25000

(2 linhas)

Também é possível acessar faixas retangulares arbitrárias da matriz, ou submatrizes. Uma faixa da matriz é especificada escrevendo *limite-inferior:limite-superior* para uma ou mais dimensões da matriz. Por exemplo, esta consulta mostra o primeiro item na agenda do Bill para os primeiros dois dias da semana:

```
SELECT agenda[1:2][1:1] FROM sal_emp WHERE nome = 'Bill';
```

```

          agenda
-----
{{reunião},{treinamento}}
(1 linha)
```

Também pode ser escrito

```
SELECT agenda[1:2][1] FROM sal_emp WHERE nome = 'Bill';
```

para obter o mesmo resultado. Uma operação com índices em matriz é sempre considerada como representando uma faixa da matriz, quando qualquer um dos índices estiver escrito na forma *inferior:superior*. O limite inferior igual a 1 é assumido para qualquer índice quando for especificado apenas um valor, como neste exemplo:

```
SELECT agenda[1:2][2] FROM sal_emp WHERE nome = 'Bill';
```

```

          agenda
-----
{{reunião,almoço},{treinamento,apresentação}}
(1 linha)
```

Podem ser obtidas as dimensões correntes de qualquer valor matriz através da função `array_dims`:

```
SELECT array_dims(agenda) FROM sal_emp WHERE nome = 'Carol';
```

```

array_dims
-----
[1:2][1:2]
(1 linha)
```

A função `array_dims` produz um resultado do tipo `text`, conveniente para as pessoas lerem mas, talvez, nem tão conveniente para os programas. As dimensões também podem ser obtidas através das funções `array_upper` e `array_lower`, que retornam os limites superior e inferior da dimensão especificada da matriz, respectivamente.

```
SELECT array_upper(agenda, 1) FROM sal_emp WHERE nome = 'Carol';
```

```

array_upper
-----
2
(1 linha)
```

#### 8.10.4. Modificação de matrizes

Um valor matriz pode ser inteiramente substituído utilizando:

```
UPDATE sal_emp SET pagamento_semanal = '{25000,25000,27000,27000}'
WHERE nome = 'Carol';
```

ou utilizando a sintaxe com a expressão `ARRAY`:

```
UPDATE sal_emp SET pagamento_semanal = ARRAY[25000,25000,27000,27000]
WHERE nome = 'Carol';
```

Também pode ser atualizado um único elemento da matriz:

```
UPDATE sal_emp SET pagamento_semanal[4] = 15000
```

```
WHERE nome = 'Bill';
```

ou pode ser atualizada uma faixa da matriz:

```
UPDATE sal_emp SET pagamento_semanal[1:2] = '{27000,27000}'
WHERE nome = 'Carol';
```

Um valor matriz armazenado pode ser ampliado fazendo atribuição a um elemento adjacente aos já presentes, ou fazendo atribuição a uma faixa que é adjacente ou se sobrepõe aos dados já presentes. Por exemplo, se a matriz `minha_matriz` possui atualmente quatro elementos, esta matriz terá cinco elementos após uma atualização que faça uma atribuição a `minha_matriz[5]`. Atualmente, as ampliações desta maneira somente são permitidas para matrizes unidimensionais, não sendo permitidas em matrizes multidimensionais.

A atribuição de faixa de matriz permite a criação de matrizes que não utilizam índices baseados em um. Por exemplo, pode ser feita a atribuição `minha_matriz[-2:7]` para criar uma matriz onde os valores dos índices variam de -2 a 7.

Também podem ser construídos novos valores matriz utilizando o operador de concatenação `||`.

```
SELECT ARRAY[1,2] || ARRAY[3,4];
```

```
?column?
-----
{1,2,3,4}
(1 linha)
```

```
SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
```

```
?column?
-----
{{5,6},{1,2},{3,4}}
(1 linha)
```

O operador de concatenação permite colocar um único elemento no início ou no fim de uma matriz unidimensional. Aceita, também, duas matrizes  $N$ -dimensionais, ou uma matriz  $N$ -dimensional e outra  $N+1$ -dimensional.

Quando é colocado um único elemento no início de uma matriz unidimensional, o resultado é uma matriz com o limite inferior do índice igual ao limite inferior do índice do operando à direita, menos um. Quando um único elemento é colocado no final de uma matriz unidimensional, o resultado é uma matriz mantendo o limite inferior do operando à esquerda. Por exemplo:

```
SELECT ARRAY[2,3];
```

```
array
-----
{2,3}
(1 linha)
```

```
SELECT array_dims(ARRAY[2,3]);
```

```
array_dims
-----
[1:2]
(1 linha)
```

```
-- Adicionar no início da matriz
```

```
SELECT 1 || ARRAY[2,3];
```

```
?column?
-----
{1,2,3}
(1 linha)
```

```
SELECT array_dims(1 || ARRAY[2,3]);
```



```

array_dims
-----
[0:2]
(1 linha)

-- Adicionar no final da matriz

```

```

SELECT ARRAY[1,2] || 3;

```

```

?column?
-----
{1,2,3}
(1 linha)

```

```

SELECT array_dims(ARRAY[1,2] || 3);

```

```

array_dims
-----
[1:3]
(1 linha)

```

Quando duas matrizes com o mesmo número de dimensões são concatenadas, o resultado mantém o limite inferior do índice da dimensão externa do operando à esquerda. O resultado é uma matriz contendo todos os elementos do operando à esquerda seguido por todos os elementos do operando à direita. Por exemplo:

```

-- Concatenação de matrizes unidimensionais

```

```

SELECT ARRAY[1,2] || ARRAY[3,4,5];

```

```

?column?
-----
{1,2,3,4,5}
(1 linha)

```

```

SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);

```

```

array_dims
-----
[1:5]
(1 linha)

```

```

-- Concatenação de matrizes bidimensionais

```

```

SELECT ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]];

```

```

?column?
-----
{{1,2},{3,4},{5,6},{7,8},{9,0}}
(1 linha)

```

```

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);

```

```

array_dims
-----
[1:5][1:2]
(1 linha)

```

Quando uma matriz  $N$ -dimensional é colocada no início ou no final de uma matriz  $N+1$ -dimensional, o resultado é análogo ao caso da matriz elemento acima. Cada submatriz  $N$ -dimensional se torna essencialmente um elemento da dimensão externa da matriz  $N+1$ -dimensional. Por exemplo:

```

-- Exemplo de matriz unidimensional concatenada com matriz bidimensional

```

```
SELECT ARRAY[1,2] || ARRAY[[3,4],[5,6]];
```

```
?column?
```

```
-----
{{1,2},{3,4},{5,6}}
(1 linha)
```

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
```

```
array_dims
-----
[0:2][1:2]
(1 linha)
```

```
-- Exemplo de matriz bidimensional concatenada com matriz tridimensional (N. do T.)
```

```
SELECT ARRAY[[-1,-2],[-3,-4]];
```

```
array
-----
{{-1,-2},{-3,-4}}
(1 linha)
```

```
SELECT array_dims(ARRAY[[-1,-2],[-3,-4]]);
```

```
array_dims
-----
[1:2][1:2]
(1 linha)
```

```
SELECT ARRAY[[[5,6],[7,8]],[[9,10],[11,12]],[[13,14],[15,16]]];
```

```
array
-----
{{{5,6},{7,8}},{9,10},{11,12}},{13,14},{15,16}}}
(1 linha)
```

```
SELECT array_dims(ARRAY[[[5,6],[7,8]],[[9,10],[11,12]],[[13,14],[15,16]]]);
```

```
array_dims
-----
[1:3][1:2][1:2]
(1 linha)
```

```
SELECT ARRAY[[-1,-2],[-3,-4]] ||
        ARRAY[[[5,6],[7,8]],[[9,10],[11,12]],[[13,14],[15,16]]];
```

```
?column?
```

```
-----
{{{{-1,-2},{-3,-4}},{5,6},{7,8}},{9,10},{11,12}},{13,14},{15,16}}}}
(1 linha)
```

```
SELECT array_dims(ARRAY[[-1,-2],[-3,-4]] ||
        ARRAY[[[5,6],[7,8]],[[9,10],[11,12]],[[13,14],[15,16]]]);
```

```
array_dims
-----
[0:3][1:2][1:2]
(1 linha)
```

```
SELECT ARRAY[[5,6],[7,8]],[9,10],[11,12],[[13,14],[15,16]] ||
        ARRAY[[-1,-2],[-3,-4]];
```

```
      ?column?
```

```
-----
{{5,6},{7,8}},{9,10},{11,12}},{13,14},{15,16}},{-1,-2},{-3,-4}}
(1 linha)
```

```
SELECT array_dims(ARRAY[[5,6],[7,8]],[9,10],[11,12],[[13,14],[15,16]] ||
                  ARRAY[[-1,-2],[-3,-4]]);
```

```
array_dims
```

```
-----
[1:4][1:2][1:2]
(1 linha)
```

Uma matriz também pode ser construída utilizando as funções `array_prepend`, `array_append` e `array_cat`. As duas primeiras suportam apenas matrizes unidimensionais, mas `array_cat` suporta matrizes multidimensionais. Deve ser observado que é preferível utilizar o operador de concatenação mostrado acima, em vez de usar diretamente estas funções. Na verdade, estas funções têm seu uso principal na implementação do operador de concatenação. Entretanto, podem ser úteis na criação de agregações definidas pelo usuário. Alguns exemplos:

```
SELECT array_prepend(1, ARRAY[2,3]);
```

```
array_prepend
```

```
-----
{1,2,3}
(1 linha)
```

```
SELECT array_append(ARRAY[1,2], 3);
```

```
array_append
```

```
-----
{1,2,3}
(1 linha)
```

```
SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
```

```
array_cat
```

```
-----
{1,2,3,4}
(1 linha)
```

```
SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
```

```
array_cat
```

```
-----
{{1,2},{3,4},{5,6}}
(1 linha)
```

```
SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
```

```
array_cat
```

```
-----
{{5,6},{1,2},{3,4}}
```

### 8.10.5. Procura em matrizes

Para procurar um valor em uma matriz deve ser verificado cada valor da matriz. Pode ser feito à mão, se for conhecido o tamanho da matriz: Por exemplo:

```
SELECT * FROM sal_emp WHERE pagamento_semanal[1] = 10000 OR
                        pagamento_semanal[2] = 10000 OR
                        pagamento_semanal[3] = 10000 OR
                        pagamento_semanal[4] = 10000;
```

Entretanto, em pouco tempo se torna entediante para matrizes grandes, e não servirá se a matriz for de tamanho desconhecido. Um método alternativo está descrito na Seção 9.17. A consulta acima pode ser substituída por:

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pagamento_semanal);
```

Além disso, podem ser encontradas as linhas onde a matriz possui todos os valores iguais a 10000 com:

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pagamento_semanal);
```

**Dica:** Matrizes não são conjuntos; a procura por determinados elementos da matriz pode ser um sinal de um banco de dados mal projetado. Considere a utilização de uma outra tabela, com uma linha para cada item que seria um elemento da matriz. Assim é mais fácil procurar e, provavelmente, vai se comportar melhor com um número grande de elementos.

### 8.10.6. Sintaxe de entrada e de saída das matrizes

A representação textual externa de um valor matriz é formada por itens que são interpretados de acordo com as regras de conversão de I/O para o tipo do elemento da matriz, mais os adornos que indicam a estrutura da matriz. Estes adornos consistem em chaves ({ e }) em torno do valor matriz, mais os caracteres delimitadores entre os itens adjacentes. O caractere delimitador geralmente é a vírgula (,), mas pode ser outro: é determinado pela definição de `typdelim` para o tipo do elemento da matriz (Entre os tipos de dado padrão fornecidos na distribuição do PostgreSQL o tipo `box` utiliza o ponto-e-vírgula (;), mas todos os outros utilizam a vírgula). Em uma matriz multidimensional cada dimensão (linha, plano, cubo, etc.) recebe seu nível próprio de chaves, e os delimitadores devem ser escritos entre entidades de chaves adjacentes do mesmo nível.

A rotina de saída de matriz coloca aspas em torno dos valores dos elementos caso estes sejam cadeias de caracteres vazias, ou se contenham chaves, caracteres delimitadores, aspas, contrabarras, ou espaços em branco. Aspas e contrabarras incorporadas aos valores dos elementos recebem o escape de contrabarra. No caso dos tipos de dado numéricos é seguro assumir que as aspas nunca vão estar presentes, mas para tipos de dado textuais deve-se estar preparado para lidar tanto com a presença quanto com a ausência das aspas (Esta é uma mudança de comportamento com relação às versões do PostgreSQL anteriores a 7.2).

Por padrão, o limite inferior do valor do índice de cada dimensão da matriz é definido como um. Se alguma das dimensões da matriz tiver um limite inferior diferente de um, um adorno adicional indicando as verdadeiras dimensões da matriz precede o adorno da estrutura da matriz. Este adorno é composto por colchetes ([ ]) em torno de cada limite inferior e superior da dimensão da matriz, com o caractere delimitador dois-pontos (:) entre estes. O adorno de dimensão da matriz é seguido pelo sinal de igual (=). Por exemplo:

```
SELECT 1 || ARRAY[2,3] AS array;
```

```
      array
-----
[0:2]={1,2,3}
(1 linha)
```

```
SELECT ARRAY[1,2] || ARRAY[[3,4]] AS array;
```

```
      array
-----
[0:1][1:2]={ {1,2}, {3,4} }
(1 linha)
```

Esta sintaxe também pode ser utilizada para especificar índices de matriz não padrão em um literal matriz. Por exemplo:

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;
```

```
e1 | e2
----+----
 1 |  6
(1 linha)
```

Conforme mostrado anteriormente, ao escrever um valor matriz pode-se colocar aspas em torno de qualquer elemento individual da matriz. Isto *deve* ser feito se o valor do elemento puder, de alguma forma, confundir o analisador de valor matriz. Por exemplo, os elementos contendo chaves, vírgulas (ou qualquer que seja o caractere delimitador), aspas, contrabarras ou espaços em branco na frente ou atrás devem estar entre aspas. Para colocar aspas ou contrabarras no valor entre aspas do elemento da matriz, estes devem ser precedidos por uma contrabarra. Como alternativa, pode ser utilizado o escape de contrabarra para proteger qualquer caractere de dado que seria de outra forma considerado como sintaxe da matriz.

Podem ser escritos espaços em branco antes do abre chaves ou após o fecha chaves. Também podem ser escritos espaços em branco antes ou depois de qualquer item individual cadeia de caracteres. Em todos estes casos os espaços em branco são ignorados. Entretanto, espaços em branco dentro de elementos entre aspas, ou envoltos nos dois lados por caracteres de um elemento que não são espaços em branco, não são ignorados.

**Nota:** Lembre-se que o que se escreve em um comando SQL é interpretado primeiro como um literal cadeia de caracteres e, depois, como uma matriz. Isto duplica o número de contrabarras necessárias. Por exemplo, para inserir um valor matriz do tipo `text` contendo uma contrabarra e uma aspa, deve ser escrito

```
INSERT ... VALUES ('{"\\", "\\"}');
```

O processador de literais cadeias de caracteres remove um nível de contrabarras, portanto o que chega para o analisador de valor matriz se parece com `{"\\", "\\"}.` Por sua vez, as cadeias de caracteres introduzidas na rotina de entrada do tipo de dado `text` se tornam `\` e `"`, respectivamente (Se estivéssemos trabalhando com um tipo de dado cuja rotina de entrada também tratasse as contrabarras de forma especial como, por exemplo, `bytea`, seriam necessárias oito contrabarras no comando para obter uma contrabarra armazenada no elemento da matriz). Pode ser utilizada a delimitação por cifrão (`dollar quoting`) (consulte a Seção 4.1.2.2) para evitar a necessidade de duplicar as contrabarras.

**Dica:** Ao se escrever valores matrizes nos comandos SQL, geralmente é mais fácil trabalhar com a sintaxe do construtor de `ARRAY` (consulte a Seção 4.2.10) do que com a sintaxe do literal cadeia de caracteres. Em `ARRAY`, os valores dos elementos individuais são escritos da mesma maneira como seriam escritos caso não fossem membros de uma matriz.

## 8.11. Tipos compostos

O *tipo composto* descreve a estrutura de uma linha ou registro; essencialmente, é apenas uma lista de nomes de campos com seus tipos de dado. O PostgreSQL permite que os valores de tipo composto sejam utilizados de muitas maneiras idênticas às que os tipos simples podem ser utilizados. Por exemplo, uma coluna de uma tabela pode ser declarada como sendo de um tipo composto.

### 8.11.1. Declaração de tipos compostos

Abaixo seguem dois exemplos simples definindo tipos compostos:

```
CREATE TYPE complexo AS (
    r      double precision,
    i      double precision
);

CREATE TYPE catalogo AS (
    nome          text,
    id_fornecedor integer,
    preco         numeric
);
```

A sintaxe pode ser comparada a do comando `CREATE TABLE`, exceto que somente podem ser especificados os nomes e tipos dos campos; atualmente não pode ser incluída nenhuma restrição (como `NOT NULL`). Deve ser observado que a

palavra chave AS é essencial; sem esta, o sistema imagina que está lidando com um tipo bem diferente de comando CREATE TYPE, e mostra erros de sintaxe bem estranhos.

Após definir os tipos, estes podem ser utilizados para criar tabelas:

```
CREATE TABLE estoque (
    item        catalogo,
    contador integer
);

INSERT INTO estoque VALUES (ROW('dados de pano', 42, 1.99), 1000);
```

ou funções:

```
CREATE FUNCTION preco_quantidade(catalogo, integer) RETURNS numeric
AS 'SELECT $1.preco * $2' LANGUAGE SQL;

SELECT preco_quantidade(item, 10) FROM estoque;
```

Sempre que uma tabela é criada também é criado, automaticamente, um tipo composto com o mesmo nome da tabela para representar o tipo linha da tabela.<sup>21</sup> Por exemplo, se tivéssemos declarado

```
CREATE TABLE catalogo (
    nome            text,
    id_fornecedor   integer REFERENCES fornecedores,
    preco           numeric CHECK (preco > 0)
);
```

teria sido criado como subproduto o mesmo tipo composto catalogo mostrado acima, podendo ser utilizado conforme mostrado anteriormente. Entretanto, deve ser observada uma restrição importante da implementação corrente: uma vez que não há nenhuma restrição associada ao tipo composto, a restrição mostrada na definição da tabela *não se aplica* aos valores do tipo composto fora da tabela (Uma forma parcial de evitar este problema é utilizar tipos domínios como membros dos tipos compostos).

### 8.11.2. Entrada de valor composto

Para escrever um valor composto como uma constante literal, os valores do campo devem ser envoltos por parênteses e separados por vírgulas. Podem ser colocadas aspas em torno de qualquer valor do campo, sendo obrigatório se o valor contiver vírgulas ou parênteses (Abaixo são mostrados mais detalhes). Portanto, o formato geral de uma constante composta é o seguinte:

```
'( val1 , val2 , ... )'
```

Por exemplo,

```
'("dados de pano",42,1.99)'
```

é um valor válido para o tipo catalogo definido acima. Para tornar o campo nulo, não deve ser escrito nenhum caractere na sua posição na lista. Por exemplo, esta constante especifica um terceiro campo nulo:

```
'("dados de pano",42,)'
```

Se, em vez de nulo, for desejada uma cadeia de caracteres vazia, devem ser escritas duas aspas:

```
'( "",42, )'
```

Neste caso, o primeiro campo é uma cadeia de caracteres vazia não-nula, e o terceiro campo é nulo.

(Estas constantes são, na verdade, apenas um caso especial do tipo genérico de constantes mostrado na Seção 4.1.2.5. Inicialmente, a constante é tratada como uma cadeia de caracteres e passada para a rotina de conversão de entrada de tipo composto (Pode ser necessária uma especificação explícita do tipo).

Também pode ser utilizada a sintaxe da expressão ROW para construir valores compostos. Na maioria dos casos, esta sintaxe é bem mais simples que a sintaxe do literal cadeia de caracteres, uma vez que não é necessário se preocupar com várias camadas de aspas. Este método já foi utilizado acima:

```
ROW('dados de pano', 42, 1.99)
ROW('', 42, NULL)
```

Desde que haja mais de um campo na expressão, a palavra chave ROW se torna opcional, permitindo simplificar como:

```
('dados de pano', 42, 1.99)
('', 42, NULL)
```

A sintaxe da expressão ROW é mostrada com mais detalhes na Seção 4.2.11.

### 8.11.3. Acesso aos tipos compostos

Para acessar um campo de uma coluna composta deve ser escrito um ponto e o nome do campo, como se faz ao selecionar um campo de uma tabela. Na verdade, é tão parecido com selecionar um campo de uma tabela que, geralmente, é necessário utilizar parênteses para não confundir o analisador. Por exemplo, selecionar alguns subcampos da tabela exemplo estoque usando algo como

```
SELECT item.nome FROM estoque WHERE item.preco > 9.99;
```

não funciona, porque o nome *item* é assumido como sendo o nome da tabela, e não o nome do campo, pelas regras de sintaxe do SQL. Devendo, então, ser escrito como mostrado abaixo

```
SELECT (item).nome FROM estoque WHERE (item).preco > 9.99;
```

ou desta forma, se também for necessário utilizar o nome da tabela (por exemplo, numa consulta com várias tabelas):

```
SELECT (estoque.item).nome FROM estoque WHERE (estoque.item).preco > 9.99;
```

Agora, como o objeto entre parênteses é interpretado corretamente como uma referência à coluna *item*, é possível selecionar um subcampo da mesma.

Ocorrem problemas de sintaxe semelhantes sempre que é selecionado um campo de um valor composto. Por exemplo, para selecionar apenas um campo do resultado de uma função que retorna um valor composto, é necessário escrever algo como:

```
SELECT (minha_funcao(...)).campo FROM ...
```

Sem os parênteses extra, provoca um erro de sintaxe.

### 8.11.4. Modificação de tipos compostos

Abaixo estão mostrados alguns exemplos da sintaxe apropriada para inserir e atualizar colunas compostas. Primeiro, são inseridas e atualizadas colunas inteiras:

```
INSERT INTO minha_tabela (coluna_complexa) VALUES((1.1,2.2));

UPDATE minha_tabela SET coluna_complexa = ROW(1.1,2.2) WHERE ...;
```

O primeiro exemplo omite ROW, enquanto o segundo exemplo não; pode ser feito de qualquer uma destas maneiras.

Os subcampos de uma coluna composta podem ser atualizados individualmente:

```
UPDATE minha_tabela SET coluna_complexa.r = (coluna_complexa).r + 1 WHERE ...;
```

Deve ser observado que não é necessário (e, na verdade, não se pode) colocar parênteses em torno do nome da coluna que aparece logo após a cláusula SET, mas são necessários parênteses ao se fazer referência à mesma coluna na expressão à direita do sinal de igual.

Também podem ser especificados subcampos como destino do INSERT:

```
INSERT INTO minha_tabela (coluna_complexa.r, coluna_complexa.i) VALUES(1.1, 2.2);
```

Caso não tivéssemos fornecido valores para todos os subcampos da coluna, os demais subcampos seriam preenchidos com o valor nulo.

### 8.11.5. Sintaxe de entrada e saída dos tipos compostos

A representação textual externa do valor composto é formada por itens que são interpretados de acordo com as regras individuais de conversão de entrada e saída do tipo de dado do campo, mais os adornos que indicam a estrutura composta. Os adornos são formados por parênteses ( ( e ) ) em torno de todo o valor, mais vírgulas ( , ) entre itens adjacentes. Os espaços em branco fora dos parênteses são ignorados, mas dentro dos parênteses são considerados parte do valor do campo, podendo ou não serem significativos dependendo das regras de conversão de entrada para o tipo de dado do campo. Por exemplo, em

```
' ( 42 ) '
```

o espaço em branco é ignorado se o tipo do campo for inteiro, mas não é ignorado se o tipo do campo for texto.

Como mostrado anteriormente, ao se escrever um valor composto podem ser escritas aspas envolvendo qualquer valor individual de campo. Isto *deve* ser feito se o valor do campo puder, de alguma forma, confundir o analisador de valores compostos. Em particular, os campos contendo parênteses, vírgulas, aspas ou contrabarras devem estar entre aspas (Além disso, um par de aspas dentro de um valor de campo envolto por aspas é assumido como representando o caractere aspas, de maneira análoga à regra para os apóstrofes nas cadeias de caracteres literais do SQL). Como alternativa, pode ser utilizado o escape de contrabarra para proteger todos os caracteres dos dados que, de outra forma, seriam assumidos como fazendo parte da sintaxe do tipo composto.

Um valor de campo inteiramente vazio (nenhum caractere entre as vírgulas ou parênteses) representa o valor nulo. Para escrever um valor que seja uma cadeia de caracteres vazia, e não o valor nulo, deve ser escrito " ".

A rotina de saída do tipo composto coloca aspas em torno dos valores dos campos caso estes sejam cadeias de caracteres vazias, ou contenham parêntese, vírgulas, aspas, contrabarras ou espaços em branco (Fazer isto para os espaços em branco não é essencial, mas melhora a legibilidade). Aspas e contrabarras incorporadas aos valores dos campos são duplicadas.

**Nota:** Deve ser lembrado que, o que se escreve em um comando SQL, é interpretado primeiro como um literal cadeia de caracteres e, depois, como um tipo composto. Isto duplica o número de contrabarras necessárias. Por exemplo, para inserir um campo do tipo `text` contendo uma contrabarra e uma aspas em um valor composto, deve ser escrito:

```
INSERT ... VALUES ('("\\""\\""\\"")');
```

O processador de literais cadeias de caracteres remove um nível de contrabarras, portanto o que chega ao analisador de valor composto se parece com ("\"\""). Por sua vez, a cadeia de caracteres introduzida na rotina de entrada do tipo de dado `text` se torna "\ (Se estivéssemos trabalhando com um tipo de dado cuja rotina de entrada também tratasse as contrabarras de forma especial como, por exemplo, `bytea`, seriam necessárias oito contrabarras no comando para obter uma contrabarra armazenada no campo composto). Pode ser utilizada a delimitação por cifrão (*dollar quoting*) (consulte a Seção 4.1.2.2) para evitar a necessidade de duplicar as contrabarras.

**Dica:** Ao se escrever valores compostos nos comandos SQL, normalmente é mais fácil trabalhar com a sintaxe do construtor `ROW` do que com a sintaxe do literal composto. Usando `ROW` os valores individuais dos campos são escritos da mesma maneira como seriam escritos se não fossem membros de um valor composto.

## 8.12. Tipos identificadores de objeto

Os identificadores de objeto (OIDs) são utilizados internamente pelo PostgreSQL como chaves primárias em várias tabelas do sistema. Além disso, uma coluna do sistema OID é adicionada às tabelas criadas pelo usuário, a menos que seja especificado `WITHOUT OIDS` na criação da tabela, ou que a variável de configuração `default_with_oids` esteja definida como falso. O tipo `oid` representa um identificador de objeto. Também existem diversos tipos aliases para `oid`: `regproc`, `regprocedure`, `regoper`, `regoperator`, `regclass`, e `regtype`. A Tabela 8-19 mostra uma visão geral.

O tipo `oid` é implementado atualmente como inteiro de quatro bytes sem sinal. Portanto, não é grande o suficiente para proporcionar unicidade para todo o banco de dados em bancos de dados grandes, ou mesmo em tabelas individuais grandes. Por isso, é desencorajada a utilização da coluna OID de uma tabela criada pelo usuário como chave primária. É melhor usar os OIDs somente para referências às tabelas do sistema.



**Nota:** Os OIDs são incluídos por padrão nas tabelas criadas pelo usuário no PostgreSQL 8.0.0. Entretanto, este comportamento provavelmente mudará em uma versão futura do PostgreSQL. Eventualmente, as tabelas criadas pelo usuário não incluirão a coluna do sistema OID, a menos que seja especificado `WITH OIDS` quando a tabela for criada, ou a variável de configuração `default_with_oids` esteja definida como verdade. Se o aplicativo requer a presença da coluna do sistema OID na tabela, deve ser especificado `WITH OIDS` na criação da tabela para garantir a compatibilidade com as versões futuras do PostgreSQL.

O tipo `oid` possui poucas operações próprias além da comparação. Pode, entretanto, ser convertido em inteiro e, então, manipulado utilizando os operadores padrão para inteiros (Tome cuidado com as possíveis confusões entre inteiros com sinal e sem sinal se isto for feito).

Os tipos aliases de `oid` não possuem operações próprias com exceção de rotinas de entrada e saída especializadas. Estas rotinas são capazes de aceitar e mostrar nomes simbólicos para objetos do sistema, em vez do valor numérico puro e simples que o tipo `oid` usaria. Os tipos aliases permitem uma procura simplificada dos valores de OID para os objetos: por exemplo, para examinar as linhas de `pg_attribute` relacionadas com a tabela `minha_tabela` pode ser escrito

```
SELECT * FROM pg_attribute WHERE attrelid = 'minha_tabela'::regclass;
```

em vez de

```
SELECT * FROM pg_attribute
WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'minha_tabela');
```

Apesar de não parecer tão ruim assim, ainda está muito simplificado. Seria necessária uma sub-seleção bem mais complicada para obter o OID correto caso existissem várias tabelas chamadas `minha_tabela` em esquemas diferentes. O conversor de entrada `regclass` trata a procura de tabela de acordo com a configuração de procura de esquema e, portanto, faz a “coisa certa” automaticamente. De forma semelhante, converter o OID da tabela para `regclass` é útil para mostrar simbolicamente o OID numérico.

**Tabela 8-19. Tipos identificadores de objetos**

Nome	Referencia	Descrição	Exemplo de valor
<code>oid</code>	qualquer um	identificador numérico de objeto	564182
<code>regproc</code>	<code>pg_proc</code>	nome de função	<code>sum</code>
<code>regprocedure</code>	<code>pg_proc</code>	função com tipos dos argumentos	<code>sum(int4)</code>
<code>regoper</code>	<code>pg_operator</code>	nome de operador	<code>+</code>
<code>regoperator</code>	<code>pg_operator</code>	operador com tipos dos argumentos	<code>*(integer,integer)</code> ou <code>-(NONE,integer)</code>
<code>regclass</code>	<code>pg_class</code>	nome da relação	<code>pg_type</code>
<code>regtype</code>	<code>pg_type</code>	nome do tipo de dado	<code>integer</code>

Todos os tipos aliases de OID aceitam nomes qualificados pelo esquema, e mostram nomes qualificados pelo esquema na saída se o objeto não puder ser encontrado no caminho de procura corrente sem que esteja qualificado. Os tipos aliases `regproc` e `regoper` somente aceitam a entrada de nomes únicos (não sobrecarregados) sendo, portanto, de uso limitado; para a maioria dos casos `regprocedure` e `regoperator` são mais apropriados. Em `regoperator`, os operadores unários são identificados escrevendo `NONE` no lugar do operando não utilizado.

Outro tipo identificador utilizado pelo sistema é o `xid`, ou identificador de transação (abreviado como `xact`). Este é o tipo de dado das colunas do sistema `xmin` e `xmax`. Os identificadores de transação são quantidades de 32 bits.

O terceiro tipo identificador é o `coid`, ou identificador de comando. Este é o tipo de dado das colunas do sistema `cmin` e `cmax`. Os identificadores de comando também são quantidades de 32 bits.

O último tipo de identificador utilizado pelo sistema é `tid`, ou identificador de tupla (identificador de linha). Este é o tipo de dado da coluna do sistema `ctid`. O identificador de tupla é um par (número do bloco, índice da tupla dentro do bloco) que identifica a posição física da linha dentro de sua tabela.

(As colunas do sistema são explicadas mais detalhadamente na Seção 5.4.)

### Exemplo 8-10. Remover as linhas duplicadas da tabela

Este exemplo utiliza os OIDs para remover as linhas duplicadas da tabela. As linhas são agrupadas por todas as colunas, exceto OID, permanecendo em cada grupo apenas a linha que possui o menor OID, que supostamente é a primeira linha do grupo que foi inserida.<sup>22</sup>

```
CREATE TEMPORARY TABLE a (c1 text, c2 text, c3 text);
INSERT INTO a VALUES ('x', 'x', 'x');
INSERT INTO a VALUES ('x', 'x', 'y'); -- 1ª duplicada (fica)
INSERT INTO a VALUES ('x', 'y', 'x');
INSERT INTO a VALUES ('x', 'x', 'y'); -- 2ª duplicada (sai)
INSERT INTO a VALUES ('x', 'x', 'y'); -- 3ª duplicada (sai)
INSERT INTO a VALUES ('x', 'y', 'y');
INSERT INTO a VALUES ('y', 'y', 'y'); -- 1ª duplicada (fica)
INSERT INTO a VALUES ('y', 'y', 'y'); -- 2ª duplicada (sai)
```

```
SELECT oid, a.* FROM a;
```

oid	c1	c2	c3
1665665	x	x	x
1665666	x	x	y
1665667	x	y	x
1665668	x	x	y
1665669	x	x	y
1665670	x	y	y
1665671	y	y	y
1665672	y	y	y

(8 linhas)

```
DELETE FROM a WHERE oid NOT IN
(SELECT min(oid) FROM a GROUP BY c1, c2, c3);
```

```
SELECT oid, a.* FROM a;
```

oid	c1	c2	c3
1665665	x	x	x
1665666	x	x	y
1665667	x	y	x
1665670	x	y	y
1665671	y	y	y

(5 linhas)

## 8.13. Pseudotipos

O sistema de tipos do PostgreSQL contém uma série de entradas com finalidades especiais chamadas coletivamente de *pseudotipos*. Um pseudotipo não pode ser utilizado como o tipo de dado de uma coluna, mas pode ser utilizado para declarar os tipos dos argumentos e dos resultados das funções. Cada um dos pseudotipos disponíveis é útil em situações onde o comportamento da função não corresponde a simplesmente aceitar ou retornar o valor de um tipo de dado específico do SQL. A Tabela 8-20 lista os pseudotipos existentes.

Tabela 8-20. Pseudotipos

Nome	Descrição
<code>any</code>	Indica que a função recebe qualquer tipo de dado entrado.
<code>anyarray</code>	Indica que a função recebe qualquer tipo de dado <code>array</code> (Consulte a Seção 31.2.5).
<code>anyelement</code>	Indica que a função aceita qualquer tipo de dado (Consulte a Seção 31.2.5).
<code>cstring</code>	Indica que a função recebe ou retorna cadeias de caracteres C terminadas por nulo.
<code>internal</code>	Indica que a função recebe ou retorna tipos de dado internos do servidor.
<code>language_handler</code>	Um tratador de chamada de linguagem procedural é declarado como retornando o tipo <code>language_handler</code> .
<code>record</code>	Identifica uma função que retorna um tipo de linha não especificado.
<code>trigger</code>	Uma função de gatilho é declarada como retornando o tipo <code>trigger</code> .
<code>void</code>	Indica que a função não retorna valor.
<code>opaque</code>	Um nome de tipo obsoleto usado no passado para todas as finalidades acima.

As funções codificadas em C (tanto nativas quanto carregadas dinamicamente) podem ser declaradas como recebendo ou retornando qualquer um destes pseudotipos de dado. É responsabilidade do autor da função garantir que a função se comporta com segurança quando é utilizado um pseudotipo como tipo do argumento.

As funções codificadas em linguagens procedurais podem utilizar somente os pseudotipos permitidos pela sua linguagem de implementação. Atualmente, todas as linguagens procedurais proíbem o uso de pseudotipos como tipo do argumento, permitindo apenas `void` e `record` como tipo do resultado (além de `trigger`, quando a função é utilizada como gatilho). Algumas linguagens também suportam funções polimórficas utilizando os tipos `anyarray` e `anyelement`.

O pseudotipo `internal` é utilizado para declarar funções feitas apenas para serem chamadas internamente pelo sistema de banco de dados, e não chamadas diretamente a partir de uma consulta SQL. Se a função possui ao menos um tipo de argumento `internal` então não pode ser chamada por um comando SQL. Para preservar a segurança de tipo desta restrição é importante seguir esta regra de codificação: não criar nenhuma função declarada como retornando o tipo `internal`, a não ser que haja pelo menos um argumento do tipo `internal`.

## Notas

1. O SQL suporta três modalidades de tipos de dado: *tipos de dado pré-definidos*, *tipos construídos* e *tipos definidos pelo usuário*. Os tipos pré-definidos são algumas vezes chamados de “tipos nativos”, mas não neste Padrão Internacional. Os tipos definidos pelo usuário podem ser definidos por um padrão, por uma implementação, ou por um aplicativo.

O tipo construído é especificado utilizando um dos construtores de tipo de dado do SQL: `ARRAY`, `MULTISET`, `REF` e `ROW`. O tipo construído é um tipo matriz, um tipo multi-conjunto, um tipo referência ou um tipo linha, se for especificado por `ARRAY`, `MULTISET`, `REF` e `ROW`, respectivamente. Os tipos matriz e multi-conjunto são conhecidos genericamente como tipos coleção.

(ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)

2. Todo tipo de dado inclui um valor especial, chamado de *valor nulo*, algumas vezes denotado pela palavra chave `NULL`. Este valor difere dos demais valores com relação aos seguintes aspectos.

— Uma vez que o valor nulo está presente em todo tipo de dado, o tipo de dado do valor nulo implicado pela palavra chave `NULL` não pode ser inferido; portanto `NULL` pode ser utilizado para denotar o valor nulo apenas em certos contextos, e não em todos os lugares onde um literal é permitido.

— Embora o valor nulo não seja igual a qualquer outro valor, nem seja não igual a qualquer outro valor - é *desconhecido* se é igual ou não a qualquer outro valor - em alguns contextos, valores nulos múltiplos são tratados juntos; por exemplo, a <cláusula group by> trata todos os valores nulos juntos.

(ISO-ANSI Working Draft) Framework (SQL/Framework), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)

3. `literal` — um valor usado exatamente da forma como é visto. Por exemplo, o número 25 e a cadeia de caracteres "Alô" são ambos literais. Os literais podem ser utilizados em expressões, e podem ser atribuídos literais para constantes ou variáveis no Visual Basic. Microsoft Glossary for Business Users (<http://www.microsoft.com/atwork/glossary.msp>) (N. do T.)
4. `deprecated` — Dito de um programa ou funcionalidade que é considerada em obsolescência e no processo de ter sua utilização gradualmente interrompida, geralmente em favor de uma determinada substituição. As funcionalidades em obsolescência podem, infelizmente, demorar muitos anos para desaparecer. The Jargon File (<http://www.catb.org/~esr/jargon/html/D/deprecated.html>) (N. do T.)
5. Padrão americano: o ponto, e não a vírgula, separando a parte fracionária. (N. do T.)
6. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
7. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
8. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
9. No SQL Server o operador `||` foi substituído pelo operador `+`.
10. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
11. `LATIN1` é a forma de codificação de caracteres especificada na ISO 8859-1. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
12. GMT — Greenwich Mean Time ( Hora média de Greenwich); UTC - Tempo Universal Coordenado ( Hora adotada por todos os países que substituiu o GMT a partir de 1972). Carimbo do Tempo, Como Funciona ? ([http://pcdsh01.on.br/Carimbo\\_ComoFunc.htm](http://pcdsh01.on.br/Carimbo_ComoFunc.htm))
13. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
14. DST — “Daylight Saving Time”, também conhecido como Horário de Verão. O DST é utilizado em muitos países como ajuste dos relógios locais, para tirar vantagem da luz natural existente durante os meses de verão. (N. do T.)
15. Dia Juliano — É obtido pela contagem de dias a partir de um ponto inicial ao meio dia em Janeiro 4713 B.C. (Dia Juliano Zero). Uma forma de informar que dia é, com a menor ambiguidade possível. ( Este sistema foi criado por Joseph Justus Scaliger, (1540 a 1609), que escolheu seu início ao meio dia em 01 de janeiro de 4713 BC, ano bissexto, ano de indicação (ano no qual, pela lei Romana, se fazia o censo das propriedades e dos indivíduos), sendo também domingo, com lua nova. — Asimov, Isaac ?"Counting the Eons") Dia Juliano (<http://pcdsh01.on.br/GloTerTec.htm#DiaJul>)
16. Os sistemas gerenciadores de banco de dados SQL Server 2000, Oracle 10g e DB2 8.1 não implementam o tipo de dado `boolean`. (N. do T.)
17. O tipo de dado booleano contém os valores verdade distintos *Verdade* e *Falso*. A menos que seja proibido pela restrição `NOT NULL`, o tipo de dado booleano também suporta o valor verdade *Desconhecido*. Esta especificação não faz distinção entre o valor nulo do tipo de dado booleano e o valor verdade “Desconhecido” resultado de um predicado, condição de procura ou expressão de valor booleana do SQL; podem ser trocados um pelo outro significando exatamente a mesma coisa. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
18. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
19. O CIDR, definido pela RFC1519, elimina o sistema de classes que determinava originalmente a parte de rede de um endereço IP. Como a sub-rede, da qual é uma extensão direta, ele conta com uma máscara de rede explícita para definir o limite entre as partes de rede e de hospedeiro de um endereço. Manual de Administração do Sistema Unix - Evi Nemeth e outros - Bookman. (N. do T.)
20. Forma abreviada de endereço de Controle de Acesso à Mídia (*Media Access Control*), um endereço de hardware que identifica unicamente cada nó da rede. Webopedia ([http://www.webopedia.com/TERM/M/MAC\\_address.html](http://www.webopedia.com/TERM/M/MAC_address.html)) (N. do T.)
21. As linhas da tabela possuem um tipo, chamado “tipo linha”; todas as linhas da tabela possuem o mesmo tipo linha, que também é o tipo linha da tabela.

Um tipo linha é uma sequência de um ou mais pares (nome de campo, tipo de dado), conhecido como campos. O valor do tipo linha consiste de um valor para cada um de seus campos.

(ISO-ANSI Working Draft) Framework (SQL/Framework), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)

22. Exemplo escrito pelo tradutor, não fazendo parte do manual original.

# Capítulo 9. Funções e Operadores

O PostgreSQL fornece um grande número de funções e operadores para os tipos de dado nativos. Os usuários também podem definir suas próprias funções e operadores, conforme descrito na Parte V. Os comandos `\df` e `\do` do `psql` podem ser utilizados para mostrar a lista de todas as funções e operadores disponíveis, respectivamente.

Havendo preocupação quanto à portabilidade, deve-se ter em mente que a maioria das funções e operadores descritos neste capítulo, com exceção dos operadores mais triviais de aritmética e de comparação, além de algumas funções indicadas explicitamente, não são especificadas pelo padrão SQL. Algumas das funcionalidades estendidas estão presentes em outros sistemas gerenciadores de banco de dados SQL e, em muitos casos, estas funcionalidades são compatíveis e consistentes entre as várias implementações.

## 9.1. Operadores lógicos

Estão disponíveis os operadores lógicos habituais:

AND  
OR  
NOT

O SQL utiliza a lógica booleana de três valores, onde o valor nulo representa o “desconhecido”. Devem ser observadas as seguintes tabelas verdade:

<b>a</b>	<b>b</b>	<b>a AND b</b>	<b>a OR b</b>
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

<b>a</b>	<b>NOT a</b>
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Os operadores `AND` e `OR` são comutativos, ou seja, pode-se trocar a ordem dos operandos esquerdo e direito sem afetar o resultado. Consulte a Seção 4.2.12 para obter informações adicionais sobre a ordem de avaliação das subexpressões.

## 9.2. Operadores de comparação

Estão disponíveis os operadores de comparação habituais, conforme mostrado na Tabela 9-1.

Tabela 9-1. Operadores de comparação

Operador	Descrição
<	menor
>	maior
<=	menor ou igual
>=	maior ou igual
=	igual
<> ou !=	diferente

**Nota:** O operador != é convertido em <> no estágio de análise. Não é possível implementar os operadores != e <> realizando operações diferentes.

Os operadores de comparação estão disponíveis para todos os tipos de dado onde fazem sentido. Todos os operadores de comparação são operadores binários, que retornam valores do tipo `boolean`; expressões como `1 < 2 < 3` não são válidas (porque não existe o operador `<` para comparar um valor booleano com 3).

Além dos operadores de comparação, está disponível a construção especial `BETWEEN`.

`a BETWEEN x AND y`

equivale a

`a >= x AND a <= y`

Analogamente,

`a NOT BETWEEN x AND y`

equivale a

`a < x OR a > y`

Não existe diferença entre as duas formas, além dos ciclos de CPU necessários para reescrever a primeira forma na segunda internamente.

Para verificar se um valor é nulo ou não, são usadas as construções

`expressão IS NULL`

`expressão IS NOT NULL`

ou às construções equivalentes, mas fora do padrão,

`expressão ISNULL`

`expressão NOTNULL`

Não deve ser escrito `expressão = NULL`, porque `NULL` não é “igual a” `NULL` (O valor nulo representa um valor desconhecido, e não se pode saber se dois valores desconhecidos são iguais). Este comportamento está de acordo com o padrão SQL.

**Dica:** Alguns aplicativos podem (incorretamente) esperar que `expressão = NULL` retorne verdade se o resultado da `expressão` for o valor nulo. É altamente recomendado que estes aplicativos sejam modificadas para ficarem em conformidade com o padrão SQL. Entretanto, se isto não puder ser feito, está disponível a variável de configuração `transform_null_equals`. Quando `transform_null_equals` está habilitada, o PostgreSQL converte as cláusulas `x = NULL` em `x IS NULL`. Este foi o comportamento padrão do PostgreSQL nas versões de 6.5 a 7.1.

O resultado dos operadores de comparação comuns é nulo (significando “desconhecido”), quando algum dos operandos é nulo. Outra forma de fazer comparação é com a construção `IS DISTINCT FROM`:

`expressão IS DISTINCT FROM expressão`

Para expressões não-nulas é o mesmo que o operador `<>`. Entretanto, quando as duas expressões são nulas retorna falso, e quando apenas uma expressão é nula retorna verdade. Portanto, atua efetivamente como se nulo fosse um valor de dado normal, em vez de “desconhecido”.

Os valores booleanos também podem ser testados utilizando as construções

`expressão IS TRUE`  
`expressão IS NOT TRUE`  
`expressão IS FALSE`  
`expressão IS NOT FALSE`  
`expressão IS UNKNOWN`  
`expressão IS NOT UNKNOWN`

Estas formas sempre retornam verdade ou falso, e nunca o valor nulo, mesmo quando o operando é nulo. A entrada nula é tratada como o valor lógico “desconhecido”. Deve ser observado que `IS UNKNOWN` e `IS NOT UNKNOWN` são efetivamente o mesmo que `IS NULL` e `IS NOT NULL`, respectivamente, exceto que a expressão de entrada deve ser do tipo booleana.

### 9.3. Funções e operadores matemáticos

São fornecidos operadores matemáticos para muitos tipos de dado do PostgreSQL. Para os tipos sem as convenções matemáticas habituais para todas as permutações possíveis (por exemplo, os tipos de data e hora), o comportamento real é descrito nas próximas seções.

A Tabela 9-2 mostra os operadores matemáticos disponíveis.

**Tabela 9-2. Operadores matemáticos**

Operador	Descrição	Exemplo	Resultado
+	adição	<code>2 + 3</code>	5
-	subtração	<code>2 - 3</code>	-1
*	multiplicação	<code>2 * 3</code>	6
/	divisão (divisão inteira trunca o resultado)	<code>4 / 2</code>	2
%	módulo (resto)	<code>5 % 4</code>	1
^	exponenciação	<code>2.0 ^ 3.0</code>	8
/	raiz quadrada	<code> / 25.0</code>	5
/	raiz cúbica	<code>  / 27.0</code>	3
!	fatorial	<code>5 !</code>	120
!!	fatorial (operador de prefixo)	<code>!! 5</code>	120
@	valor absoluto	<code>@ -5.0</code>	5
&	AND bit a bit	<code>91 &amp; 15</code>	11
	OR bit a bit	<code>32   3</code>	35
#	XOR bit a bit	<code>17 # 5</code>	20
~	NOT bit a bit	<code>~1</code>	-2
<<	deslocamento à esquerda bit a bit	<code>1 &lt;&lt; 4</code>	16
>>	deslocamento à direita bit a bit	<code>8 &gt;&gt; 2</code>	2



Os operadores bit a bit <sup>1</sup> trabalham somente em tipos de dado inteiros, enquanto os demais estão disponíveis para todos os tipos de dado numéricos. Os operadores bit a bit também estão disponíveis para os tipos cadeia de bits `bit` e `bit varying`, conforme mostrado na Tabela 9-12.

A Tabela 9-3 mostra as funções matemáticas disponíveis. Nesta tabela “dp” significa `double precision`. Muitas destas funções são fornecidas em várias formas, com diferentes tipos de dado dos argumentos. Exceto onde estiver indicado, todas as formas das funções retornam o mesmo tipo de dado de seu argumento. As funções que trabalham com dados do tipo `double precision` são, em sua maioria, implementadas usando a biblioteca C do sistema hospedeiro; a precisão e o comportamento em casos limites podem, portanto, variar dependendo do sistema hospedeiro.

**Tabela 9-3. Funções matemáticas**

Função	Tipo retornado	Descrição	Exemplo	Resultado
<code>abs(x)</code>	(o mesmo de <code>x</code> )	valor absoluto	<code>abs(-17.4)</code>	17.4
<code>cbrt(dp)</code>	dp	raiz cúbica	<code>cbrt(27.0)</code>	3
<code>ceil(dp ou numeric)</code>	(o mesmo da entrada)	o menor inteiro não menor que o argumento	<code>ceil(-42.8)</code>	-42
<code>ceiling(dp ou numeric)</code>	(o mesmo da entrada)	o menor inteiro não menor que o argumento (o mesmo que <code>ceil</code> )	<code>ceiling(-95.3)</code>	-95
<code>degrees(dp)</code>	dp	radianos para graus	<code>degrees(0.5)</code>	28.6478897565412
<code>exp(dp ou numeric)</code>	(o mesmo da entrada)	exponenciação	<code>exp(1.0)</code>	2.71828182845905
<code>floor(dp ou numeric)</code>	(o mesmo da entrada)	o maior inteiro não maior que o argumento	<code>floor(-42.8)</code>	-43
<code>ln(dp ou numeric)</code>	(o mesmo da entrada)	logaritmo natural	<code>ln(2.0)</code>	0.693147180559945
<code>log(dp ou numeric)</code>	(o mesmo da entrada)	logaritmo na base 10	<code>log(100.0)</code>	2
<code>log(b numeric, x numeric)</code>	numeric	logaritmo na base b	<code>log(2.0, 64.0)</code>	6.0000000000
<code>mod(y, x)</code>	(same as argument types)	resto de <code>y/x</code>	<code>mod(9, 4)</code>	1
<code>pi()</code>	dp	constante “ $\pi$ ”	<code>pi()</code>	3.14159265358979
<code>power(a dp, b dp)</code>	dp	a elevado a b	<code>power(9.0, 3.0)</code>	729
<code>power(a numeric, b numeric)</code>	numeric	a elevado a b	<code>power(9.0, 3.0)</code>	729
<code>radians(dp)</code>	dp	graus para radianos	<code>radians(45.0)</code>	0.785398163397448
<code>random()</code>	dp	valor randômico entre 0.0 e 1.0	<code>random()</code>	

Função	Tipo retornado	Descrição	Exemplo	Resultado
<code>round(dp ou numeric)</code>	(o mesmo da entrada)	arredondar para o inteiro mais próximo	<code>round(42.4)</code>	42
<code>round(v numeric, s integer)</code>	numeric	arredondar para s casas decimais	<code>round(42.4382, 2)</code>	42.44
<code>setseed(dp)</code>	integer	define a semente para as próximas chamadas a <code>random()</code>	<code>setseed(0.54823)</code>	1177314959
<code>sign(dp ou numeric)</code>	(o mesmo da entrada)	sinal do argumento (-1, 0, +1)	<code>sign(-8.4)</code>	-1
<code>sqrt(dp ou numeric)</code>	(o mesmo da entrada)	raiz quadrada	<code>sqrt(2.0)</code>	1.4142135623731
<code>trunc(dp ou numeric)</code>	(o mesmo da entrada)	trunca em direção ao zero	<code>trunc(42.8)</code>	42
<code>trunc(v numeric, s integer)</code>	numeric	trunca com s casas decimais	<code>trunc(42.4382, 2)</code>	42.43
<code>width_bucket(operando numeric, b1 numeric, b2 numeric, contador integer)</code>	integer	retorna a barra à qual o operando seria atribuído, em um histograma equidepth com contador barras, um limite superior de b1, e um limite inferior de b2	<code>width_bucket(5.35, 0.024, 10.06, 5)</code>	3

A Tabela 9-4 mostra as funções trigonométricas disponíveis. Todas as funções trigonométricas recebem argumentos e retornam valores do tipo `double precision`.

**Tabela 9-4. Funções trigonométricas**

Função	Descrição
<code>acos(x)</code>	arco cosseno
<code>asin(x)</code>	arco seno
<code>atan(x)</code>	arco tangente
<code>atan2(x, y)</code>	arco tangente de $x/y$
<code>cos(x)</code>	cosseno
<code>cot(x)</code>	cotangente
<code>sin(x)</code>	seno
<code>tan(x)</code>	tangente

A Tabela 9-5 compara as funções matemáticas e trigonométricas do Oracle (<http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/functions2a.htm>), do SQL Server ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts\\_fa-fz\\_24c3.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts_fa-fz_24c3.asp)), do DB2 (<https://aurora.vcu.edu/db2help/db2s0/ch2func.htm#HDRCH2FUNC>) e do PostgreSQL 8.0.0.

**Nota:** Tabela escrita pelo tradutor, não fazendo parte do manual original.

**Tabela 9-5. Funções matemáticas e trigonométricas do Oracle 9i, do SQL Server 2000, do DB2 8.1 e do PostgreSQL 8.0.0**

Função	Oracle 9i <sup>a</sup>	SQL Server 2000 <sup>b</sup>	DB2 8.1	PostgreSQL 8.0.0
valor absoluto	abs(n)	abs(n)	abs(n) ou absval(n)	abs(x)
arco cosseno	acos(n)	acos(f)	acos(d)	acos(x)
arco seno	asin(n)	asin(f)	asin(d)	asin(x)
arco tangente de n	atan(n)	atan(f)	atan(d)	atan(x)
arco tangente de x/y	atan2(n, m)	atn2(f, f)	atan2(d, d)	atan2(x, y)
raiz cúbica	-	-	-	cbrt(dp)
menor inteiro não menor que o argumento	ceil(n)	ceiling(n)	ceil(n) ou ceiling(n)	ceil(dp ou numeric)
cosseno	cos(n)	cos(f)	cos(d)	cos(x)
cosseno hiperbólico	cosh(n)	-	-	-
cotangente	-	cot(f)	cot(d)	cot(x)
radianos para graus	-	degrees(n)	degrees(d)	degrees(dp)
exponenciação	exp(n)	exp(f)	exp(d)	exp(dp ou numeric)
maior inteiro não maior que o argumento	floor(n)	floor(n)	floor(n)	floor(dp ou numeric)
logaritmo natural	ln(n)	log(f)	ln(d) ou log(d)	ln(dp ou numeric)
logaritmo, qualquer base	log(m, n)	-	-	log(b numeric, x numeric)
logaritmo, base 10	log(10, n)	log10(f)	log10(d)	log(dp ou numeric)
módulo (resto)	mod(m, n)	dividendo % divisor	mod(n, n)	mod(y, x)
constante $\pi$	-	pi()	-	pi()
potenciação	power(m, n)	power(n, y)	power(n, n)	pow(a dp, b dp) e pow(a numeric, b numeric)
radianos	-	radians(n)	radians(d)	radians(dp)
número randômico	-	rand()	rand()	random()
arredondar para o inteiro mais próximo	round(n)	round(n, 0)	round(n, 0)	round(dp ou numeric)
arredondar para s casas decimais	round(n [, s integer])	round(n, s integer [, função])	round(n, s integer)	round(v numeric, s integer)

Função	Oracle 9i <sup>a</sup>	SQL Server 2000 <sup>b</sup>	DB2 8.1	PostgreSQL 8.0.0
define a semente para as próximas chamadas a random()	-	rand( semente )	rand( semente )	setseed( dp )
sinal do argumento (-1, 0, +1)	sign( n )	sign( n )	sign( n )	sign( dp ou numeric )
seno	sin( n )	sin( f )	sin( d )	sin( x )
seno hiperbólico	sinh( n )	-	-	-
raiz quadrada	sqrt( n )	sqrt( f )	sqrt( d )	sqrt( dp ou numeric )
tangente	tan( n )	tan( f )	tan( d )	tan( x )
tangente hiperbólica	tanh( n )	-	-	-
trunca em direção ao zero	trunc( n )	-	trunc( n, 0 )	trunc( dp ou numeric )
trunca com s casas decimais	trunc( n [, s integer] )	-	trunc( n , s integer )	trunc( v numeric, s integer )
Notas: a. Oracle 9i — As funções numéricas recebem entradas numéricas e retornam valores numéricos. A maior parte destas funções retornam valores com precisão de 38 dígitos decimais. As funções transcendentais COS, COSH, EXP, LN, LOG, SIN, SINH, SQRT, TAN e TANH têm precisão de 36 dígitos decimais. As funções transcendentais ACOS, ASIN, ATAN e ATAN2 têm precisão de 30 dígitos decimais. b. SQL Server 2000 — As funções aritméticas, tais como ABS, CEILING, DEGREES, FLOOR, POWER, RADIANS e SIGN, retornam um valor que possui o mesmo tipo de dado do valor da entrada. As funções trigonométricas e as demais funções, incluindo EXP, LOG, LOG10, SQUARE e SQRT convertem seus valores de entrada em ponto flutuante, e retornam um valor de ponto flutuante. Todas as funções matemáticas, exceto RAND, são funções determinísticas; retornam o mesmo resultado toda vez que são chamadas com um determinado conjunto de valores de entrada. RAND só é determinística quando é especificado o parâmetro semente.				

## 9.4. Funções e operadores para cadeias de caracteres

Esta seção descreve as funções e operadores disponíveis para examinar e manipular valores cadeia de caracteres. Neste contexto, cadeia de caracteres inclui todos os valores dos tipos `character`, `character varying` e `text`. A menos que seja dito o contrário, todas as funções relacionadas abaixo trabalham com todos estes tipos, mas se deve tomar cuidado com os efeitos em potencial do preenchimento automático quando for utilizado o tipo `character`. De modo geral, as funções descritas nesta seção também trabalham com dados de tipos que não são cadeias de caracteres, convertendo estes dados primeiro na representação de cadeia de caracteres. Algumas funções também existem em forma nativa para os tipos cadeia de bits.

O SQL define algumas funções para cadeias de caracteres com uma sintaxe especial, onde certas palavras chave, em vez de vírgulas, são utilizadas para separar os argumentos. Os detalhes estão na Tabela 9-6. Estas funções também são implementadas utilizando a sintaxe regular de chamada de função (Consulte a Tabela 9-7).

**Tabela 9-6. Funções e operadores SQL para cadeias de caracteres**

Função	Tipo retornado	Descrição	Exemplo	Resultado
cadeia_de_caracteres    cadeia_de_caracteres	text	Concatenação de cadeias de caracteres	'Post'    'greSQL'	PostgreSQL

Função	Tipo retornado	Descrição	Exemplo	Resultado
<code>bit_length(cadeia_de_caracteres)</code>	integer	Número de bits na cadeia de caracteres	<code>bit_length('José')</code>	32
<code>char_length(cadeia_de_caracteres)</code> OU <code>character_length(cadeia_de_caracteres)</code>	integer	Número de caracteres na cadeia de caracteres	<code>char_length('José')</code>	4
<code>convert(cadeia_de_caracteres using nome_da_conversão)</code>	text	Muda a codificação utilizando o nome de conversão especificado. As conversões podem ser definidas pelo comando <code>CREATE CONVERSION</code> . Além disso, existem alguns nomes de conversão pré-definidos. Veja na Tabela 9-8 os nomes de conversão disponíveis.	<code>convert('PostgreSQL' using iso_8859_1_to_utf_8)</code>	'PostgreSQL' na codificação Unicode (UTF-8)
<code>lower(cadeia_de_caracteres)</code>	text	Converte as letras da cadeia de caracteres em minúsculas	<code>lower('SÃO')</code>	são
<code>octet_length(cadeia_de_caracteres)</code>	integer	Número de bytes na cadeia de caracteres	<code>octet_length('José')</code>	4
<code>overlay(cadeia_de_caracteres placing cadeia_de_caracteres from integer [for integer])</code>	text	Substituir parte da cadeia de caracteres (sobreposição)	<code>overlay('Txxxxas' placing 'hom' from 2 for 4)</code>	Thomas
<code>position(substring in cadeia_de_caracteres)</code>	integer	Posição da subcadeia de caracteres	<code>position('om' in 'Thomas')</code>	3
<code>substring(cadeia_de_caracteres [from integer] [for integer])</code>	text	Extraí parte da cadeia de caracteres	<code>substring('Thomas' from 2 for 3)</code>	hom
<code>substring(cadeia_de_caracteres from padrão)</code>	text	Extraí a parte da cadeia de caracteres correspondente à expressão regular POSIX ( <a href="http://unixhelp.ed.ac.uk/CGI/man-cgi?regex+7">http://unixhelp.ed.ac.uk/CGI/man-cgi?regex+7</a> ) <sup>a</sup>	<code>substring('Thomas' from '...\$')</code>	mas
<code>substring(cadeia_de_caracteres from padrão for escape)</code>	text	Extraí a parte da cadeia de caracteres correspondente à expressão regular SQL	<code>substring('Thomas' from '%#o_a#"' for '#')</code>	oma

Função	Tipo retornado	Descrição	Exemplo	Resultado
<code>trim([leading   trailing   both] [caracteres] from cadeia_de_caracteres)</code>	text	Remove a cadeia de caracteres mais longa contendo apenas os caracteres (espaço por padrão) da extremidade inicial/final/ambas da cadeia_de_caracteres.	<code>trim(both 'x' from 'xTomxx')</code>	Tom
<code>upper( cadeia_de_caracteres)</code>	text	Converte as letras da cadeia de caracteres em maiúsculas	<code>upper('são')</code>	São
Notas: a. Nos sistemas *nix execute <code>man 7 regex</code> para ver uma descrição das expressões regulares POSIX 1003.2. (N. do T.)				

Estão disponíveis funções adicionais para manipulação de cadeias de caracteres, conforme mostrado na Tabela 9-7. Algumas delas são utilizadas internamente para implementar funções de cadeia de caracteres do padrão SQL, conforme mostrado na Tabela 9-6.

**Tabela 9-7. Outras funções para cadeia de caracteres**

Função	Tipo retornado	Descrição	Exemplo	Resultado
<code>ascii(text)</code>	integer	código ASCII do primeiro caractere do argumento	<code>ascii('x')</code>	120
<code>btrim( cadeia_de_caracteres text [, caracteres text])</code>	text	Remove a maior cadeia de caracteres contendo apenas os caracteres presentes em caracteres (espaço por padrão), do início e do fim da cadeia_de_caracteres	<code>btrim('xytrimyyx', 'xy')</code>	trim
<code>chr(integer)</code>	text	Caractere com o código ASCII fornecido	<code>chr(65)</code>	A
<code>convert(cadeia_de_caracteres text, [codificação_de_origem name,] codificação_de_destino name)</code>	text	Converte a cadeia de caracteres na codificação_de_destino. A codificação de origem é especificada por codificação_de_origem. Se a codificação_de_origem for omitida, será assumida a codificação do banco de dados.	<code>convert('texto_em_unicode', 'UNICODE', 'LATIN1')</code>	Representação na codificação ISO 8859-1 do texto_em_unicode
<code>decode(cadeia_de_caracteres text, tipo text)</code>	bytea	Decodifica os dados binários da cadeia_de_caracteres previamente codificada com <code>encode()</code> . O tipo do parâmetro é o mesmo que em <code>encode()</code> .	<code>decode('MTIzAAE=', 'base64')</code>	123\000\001
<code>encode(dados bytea, tipo text)</code>	text	Codifica dados binários na representação somente ASCII. Os tipos suportados são:	<code>encode('123\000\001', 'base64')</code>	MTIzAAE=

Função	Tipo retornado	Descrição	Exemplo	Resultado
		base64, hex e escape.		
<code>initcap(text)</code>	text	Converte a primeira letra de cada palavra em maiúscula e as demais em minúsculas. As palavras são seqüências de caracteres alfanuméricos separadas por caracteres não alfanuméricos.	<code>initcap('hi THOMAS')</code>	Hi Thomas
<code>length( cadeia_de_caracteres text)</code>	integer	Número de caracteres presentes na cadeia_de_caracteres.	<code>length('José')</code>	4
<code>lpad( cadeia_de_caracteres text, comprimento integer [, preenchimento text])</code>	text	Preenche a cadeia_de_caracteres até o comprimento adicionando os caracteres de preenchimento (espaço por padrão) à esquerda. Se a cadeia_de_caracteres for mais longa que o comprimento então é truncada (à direita).	<code>lpad('hi', 5, 'xy')</code>	xyxhi
<code>ltrim( cadeia_de_caracteres text [, caracteres text])</code>	text	Remove a cadeia de caracteres mais longa contendo apenas caracteres presentes em caracteres (espaço por padrão) do início da cadeia_de_caracteres.	<code>ltrim('zzzytrim', 'xyz')</code>	trim
<code>md5( cadeia_de_caracteres text)</code>	text	Calcula o MD5 da cadeia_de_caracteres, retornando o resultado em hexadecimal.	<code>md5('abc')</code>	900150983cd24fb0d6963f7d28e17f72
<code>pg_client_encoding()</code>	name	Nome da codificação atual do cliente	<code>pg_client_encoding()</code>	LATIN1
<code>quote_ident( cadeia_de_caracteres text)</code>	text	Retorna a cadeia de caracteres fornecida apropriadamente entre aspas, para ser utilizada como identificador na cadeia de caracteres de um comando SQL. As aspas são adicionadas somente quando há necessidade (ou seja, se a cadeia de caracteres contiver caracteres não-identificadores, ou se contiver letras maiúsculas e minúsculas). As aspas internas são devidamente duplicadas.	<code>quote_ident('Foo bar')</code>	"Foo bar"
<code>quote_literal( cadeia_de_</code>	text	Retorna a cadeia de caracteres fornecida apropriadamente	<code>quote_literal(</code>	'O'Reilly'

Função	Tipo retornado	Descrição	Exemplo	Resultado
<code>caracteres text)</code>		entre apóstrofes, para ser utilizada como literal cadeia de caracteres na cadeia de caracteres de um comando SQL. Os apóstrofes e contrabarras embutidos são devidamente duplicados.	<code>'O\'Reilly')</code>	
<code>repeat(cadeia_de_caracteres text, número integer)</code>	text	Repete a cadeia_de_caracteres pelo número de vezes especificado	<code>repeat('Pg', 4)</code>	PgPgPgPg
<code>replace(cadeia_de_caracteres text, origem text, destino text)</code>	text	Substitui todas as ocorrências na cadeia_de_caracteres, da cadeia de caracteres de origem pela cadeia de caracteres de destino.	<code>replace('abcdefabcdef', 'cd', 'XX')</code>	abXXefabXXef
<code>rpadd(cadeia_de_caracteres text, comprimento integer [, preenchimento text])</code>	text	Preenche a cadeia_de_caracteres até o comprimento anexando os caracteres de preenchimento (espaço por padrão) à direita. Se a cadeia_de_caracteres for mais longa que o comprimento, então é truncada.	<code>rpadd('hi', 5, 'xy')</code>	hixyx
<code>rtrim(cadeia_de_caracteres text [, caracteres text])</code>	text	Remove do final da cadeia_de_caracteres, a cadeia de caracteres mais longa contendo apenas os caracteres presentes em caracteres (espaço por padrão).	<code>rtrim('trimxxxx', 'x')</code>	trim
<code>split_part(cadeia_de_caracteres text, delimitador text, campo integer)</code>	text	Divide a cadeia_de_caracteres utilizando o delimitador, retornando o campo especificado (contado a partir de 1).	<code>split_part('abc~~def~~ghi', '~', 2)</code>	def
<code>strpos(cadeia_de_caracteres, caracteres)</code>	text	Posição dos caracteres especificados; o mesmo que <code>position(caracteres in cadeia_de_caracteres)</code> , mas deve ser observada a ordem invertida dos argumentos	<code>strpos('high', 'ig')</code>	2
<code>substr(cadeia_de_caracteres, origem [, contador])</code>	text	Extraí a subcadeia de caracteres caracteres; o mesmo que <code>substring(cadeia_de_cara</code>	<code>substr('alphabet', 3, 2)</code>	ph



Função	Tipo retornado	Descrição	Exemplo	Resultado
		cteres from origem for contador)		
<code>to_ascii(text [, codificação])</code>	text	Converte texto em outras codificações em ASCII <sup>a</sup>	<code>to_ascii('Consequência')</code>	Consequencia
<code>to_hex(número integer ou bigint)</code>	text	Converte o número em sua representação hexadecimal equivalente	<code>to_hex(2147483647)</code>	7fffffff
<code>translate(cadeia_de_caracteres text, origem text, destino text)</code>	text	Todo caractere da cadeia_de_caracteres que corresponde a um caractere do conjunto origem, é substituído pelo caractere correspondente do conjunto destino.	<code>translate('12345', '14', 'ax')</code>	a23x5
Notas: a. A função <code>to_ascii</code> permite apenas a conversão das codificações LATIN1, LATIN2, LATIN9 e WIN1250.				

### Exemplo 9-1. Conversão de letras minúsculas e maiúsculas acentuadas

Abaixo estão mostradas duas funções para conversão de letras. A função `maiusculas` converte letras minúsculas, com ou sem acentos, em maiúsculas, enquanto a função `minusculas` faz o contrário, ou seja, converte letras maiúsculas, com ou sem acentos em minúsculas <sup>2</sup>.

```
=> \!chcp 1252
Active code page: 1252
=> CREATE FUNCTION maiusculas(text) RETURNS text AS '
'>     SELECT translate( upper($1),
'>         text ' 'áéíóúâëîôûãõäêîöôäëïöüç' ',
'>         text ' 'ÃÊÎÕÜÄËÎÖÜÃÕÄÊÎÖÜÄËÎÖÜÇ' ')
'> ' LANGUAGE SQL STRICT;

=> SELECT maiusculas('à ação seqüência');

      maiusculas
-----
À AÇÃO SEQÜÊNCIA
(1 linha)

=> CREATE FUNCTION minusculas(text) RETURNS text AS '
'>     SELECT translate( lower($1),
'>         text ' 'ÃÊÎÕÜÄËÎÖÜÃÕÄÊÎÖÜÄËÎÖÜÇ' ',
'>         text ' 'áéíóúâëîôûãõäêîöôäëïöüç' ')
'> ' LANGUAGE SQL STRICT;

=> SELECT minusculas('À AÇÃO SEQÜÊNCIA');

      minusculas
-----
à ação seqüência
(1 linha)
```

Tabela 9-8. Conversões nativas

Nome da conversão <sup>a</sup>	Codificação de origem	Codificação de destino
ascii_to_mic	SQL_ASCII	MULE_INTERNAL
ascii_to_utf_8	SQL_ASCII	UNICODE
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf_8	BIG5	UNICODE
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf_8	EUC_CN	UNICODE
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf_8	EUC_JP	UNICODE
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf_8	EUC_KR	UNICODE
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf_8	EUC_TW	UNICODE
gb18030_to_utf_8	GB18030	UNICODE
gbk_to_utf_8	GBK	UNICODE
iso_8859_10_to_utf_8	LATIN6	UNICODE
iso_8859_13_to_utf_8	LATIN7	UNICODE
iso_8859_14_to_utf_8	LATIN8	UNICODE
iso_8859_15_to_utf_8	LATIN9	UNICODE
iso_8859_16_to_utf_8	LATIN10	UNICODE
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf_8	LATIN1	UNICODE
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf_8	LATIN2	UNICODE
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf_8	LATIN3	UNICODE
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf_8	LATIN4	UNICODE
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf_8	ISO_8859_5	UNICODE

Nome da conversão <sup>a</sup>	Codificação de origem	Codificação de destino
iso_8859_5_to_windows_1251	ISO_8859_5	WIN
iso_8859_5_to_windows_866	ISO_8859_5	ALT
iso_8859_6_to_utf_8	ISO_8859_6	UNICODE
iso_8859_7_to_utf_8	ISO_8859_7	UNICODE
iso_8859_8_to_utf_8	ISO_8859_8	UNICODE
iso_8859_9_to_utf_8	LATIN5	UNICODE
johab_to_utf_8	JOHAB	UNICODE
koi8_r_to_iso_8859_5	KOI8	ISO_8859_5
koi8_r_to_mic	KOI8	MULE_INTERNAL
koi8_r_to_utf_8	KOI8	UNICODE
koi8_r_to_windows_1251	KOI8	WIN
koi8_r_to_windows_866	KOI8	ALT
mic_to_ascii	MULE_INTERNAL	SQL_ASCII
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN
mic_to_windows_866	MULE_INTERNAL	ALT
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf_8	SJIS	UNICODE
tcvn_to_utf_8	TCVN	UNICODE
uhc_to_utf_8	UHC	UNICODE
utf_8_to_ascii	UNICODE	SQL_ASCII
utf_8_to_big5	UNICODE	BIG5

Nome da conversão <sup>a</sup>	Codificação de origem	Codificação de destino
utf_8_to_euc_cn	UNICODE	EUC_CN
utf_8_to_euc_jp	UNICODE	EUC_JP
utf_8_to_euc_kr	UNICODE	EUC_KR
utf_8_to_euc_tw	UNICODE	EUC_TW
utf_8_to_gbl8030	UNICODE	GB18030
utf_8_to_gbk	UNICODE	GBK
utf_8_to_iso_8859_1	UNICODE	LATIN1
utf_8_to_iso_8859_10	UNICODE	LATIN6
utf_8_to_iso_8859_13	UNICODE	LATIN7
utf_8_to_iso_8859_14	UNICODE	LATIN8
utf_8_to_iso_8859_15	UNICODE	LATIN9
utf_8_to_iso_8859_16	UNICODE	LATIN10
utf_8_to_iso_8859_2	UNICODE	LATIN2
utf_8_to_iso_8859_3	UNICODE	LATIN3
utf_8_to_iso_8859_4	UNICODE	LATIN4
utf_8_to_iso_8859_5	UNICODE	ISO_8859_5
utf_8_to_iso_8859_6	UNICODE	ISO_8859_6
utf_8_to_iso_8859_7	UNICODE	ISO_8859_7
utf_8_to_iso_8859_8	UNICODE	ISO_8859_8
utf_8_to_iso_8859_9	UNICODE	LATIN5
utf_8_to_johab	UNICODE	JOHAB
utf_8_to_koi8_r	UNICODE	KOI8
utf_8_to_sjis	UNICODE	SJIS
utf_8_to_tcvn	UNICODE	TCVN
utf_8_to_uhc	UNICODE	UHC
utf_8_to_windows_1250	UNICODE	WIN1250
utf_8_to_windows_1251	UNICODE	WIN
utf_8_to_windows_1256	UNICODE	WIN1256
utf_8_to_windows_866	UNICODE	ALT
utf_8_to_windows_874	UNICODE	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf_8	WIN1250	UNICODE
windows_1251_to_iso_8859_5	WIN	ISO_8859_5
windows_1251_to_koi8_r	WIN	KOI8

Nome da conversão <sup>a</sup>	Codificação de origem	Codificação de destino
windows_1251_to_mic	WIN	MULE_INTERNAL
windows_1251_to_utf_8	WIN	UNICODE
windows_1251_to_windows_866	WIN	ALT
windows_1256_to_utf_8	WIN1256	UNICODE
windows_866_to_iso_8859_5	ALT	ISO_8859_5
windows_866_to_koi8_r	ALT	KOI8
windows_866_to_mic	ALT	MULE_INTERNAL
windows_866_to_utf_8	ALT	UNICODE
windows_866_to_windows_1251	ALT	WIN
windows_874_to_utf_8	WIN874	UNICODE
Notas: a. Os nomes das conversões obedecem a um esquema de nomes padronizado: O nome oficial da codificação de origem, com todos os caracteres não alfanuméricos substituídos por sublinhado, seguido por <code>_to_</code> , seguido pelo nome da codificação de destino processado da mesma forma que o nome da codificação de origem. Portanto, os nomes podem desviar dos nomes habituais das codificações.		

### 9.4.1. Comparações entre o PostgreSQL, o Oracle, o SQL Server e o DB2

**Nota:** Seção escrita pelo tradutor, não fazendo parte do manual original.

Os exemplos abaixo comparam funções e operadores para cadeias de caracteres do Oracle (<http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/functions2a.htm>), do SQL Server ([http://msdn.microsoft.com/library/en-us/tsqlref/ts\\_fa-fz\\_7oqb.asp](http://msdn.microsoft.com/library/en-us/tsqlref/ts_fa-fz_7oqb.asp)), do DB2 (<https://aurora.vcu.edu/db2help/db2s0/ch2func.htm#HDRCH2FUNC>) e do PostgreSQL 8.0.0.

#### Exemplo 9-2. Tamanho de uma cadeia de caracteres com espaço à direita

Abaixo são mostradas consultas que retornam como resultado o tamanho de uma cadeia de caracteres com espaço à direita. Note que apenas a função `len` do SQL Server 2000 não conta o espaço à direita.

PostgreSQL 8.0.0:

```
=> SELECT length('1234567890 ');
```

```
length
-----
      11
(1 linha)
```

SQL Server 2000:

```
SELECT len('1234567890 ') AS len
```

```
len
---
10
(1 row(s) affected)
```

Oracle 10g:

```
SQL> SELECT length('1234567890 ') AS length FROM sys.dual;
```

```
LENGTH
-----
      11
```

DB2 8.1:

```
DB2SQL92> SELECT length('1234567890 ') FROM sysibm.sysdummy1;
```

```
1
-----
      11
```

### Exemplo 9-3. Concatenação de cadeias de caracteres

Abaixo são mostradas consultas que retornam como resultado a concatenação de cadeias de caracteres. Deve ser observado que o SQL Server 2000 usa o operador +, enquanto os demais usam o operador || para concatenar cadeias de caracteres. Além disso, o Oracle e o DB2 possuem a função concat, nativa, para concatenar duas cadeias de caracteres. Embora o PostgreSQL e o SQL Server não possuam a função concat nativa, esta pode ser facilmente definida neste dois produtos.

PostgreSQL 8.0.0:

```
=> SELECT 'ae' || 'io' || 'u' AS vogais;
```

```
vogais
-----
aeiou
(1 linha)
```

```
=> CREATE FUNCTION concat(text,text) RETURNS text AS '
'>     SELECT $1 || $2;
'> ' LANGUAGE SQL STRICT;
```

```
=> SELECT concat(concat('ae','io'),'u') AS vogais;
```

```
vogais
-----
aeiou
(1 linha)
```

SQL Server 2000:

```
SELECT 'ae' + 'io' + 'u' AS vogais;
```

```
vogais
-----
aeiou
(1 row(s) affected)
```

```
CREATE FUNCTION dbo.concat(@s1 varchar(64), @s2 varchar(64))
RETURNS varchar(128)
AS
BEGIN
    RETURN @s1 + @s2
END
GO
SELECT dbo.concat(dbo.concat('ae','io'),'u') AS vogais
GO
```

```
vogais
-----
aeiou
(1 row(s) affected)
```

Oracle 10g:

```
SQL> SELECT 'ae' || 'io' || 'u' AS vogais FROM sys.dual;
```

```
VOGAI
-----
aeiou
```

```
SQL> SELECT concat(concat('ae','io'),'u') AS vogais FROM sys.dual;
```

```
VOGAI
-----
aeiou
```

DB2 8.1:

```
DB2SQL92> SELECT 'ae' || 'io' || 'u' AS vogais FROM sysibm.sysdummy1;
```

```
VOGAIS
-----
aeiou
```

```
DB2SQL92> SELECT concat(concat('ae','io'),'u') AS vogais FROM sysibm.sysdummy1;
```

```
VOGAIS
-----
aeiou
```

### 9.4.2. Similaridades entre o PostgreSQL e o Oracle

**Nota:** Seção escrita pelo tradutor, não fazendo parte do manual original.

A Tabela 9-9 compara funções e operadores para cadeias de caracteres do PostgreSQL e do Oracle, através de exemplos tirados do próprio manual do Oracle ([http://download-west.oracle.com/docs/cd/B10501\\_01/server.920/a96540/functions2a.htm](http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96540/functions2a.htm)), com intuito de mostrar similaridades.

**Tabela 9-9. Funções e operadores para cadeias de caracteres do PostgreSQL e do Oracle**

PostgreSQL 8.0.0	Oracle 10g
<pre>=&gt; SELECT CHR(67)    CHR(65)    CHR(84) AS "Dog";  Dog ----- CAT</pre>	<pre>SQL&gt; SELECT CHR(67)    CHR(65)    CHR(84) "Dog" 2 FROM sys.dual;  Dog --- CAT</pre>
<pre>=&gt; SELECT INITCAP('the soap') AS "Capitals";  Capitals ----- The Soap</pre>	<pre>SQL&gt; SELECT INITCAP('the soap') "Capitals" 2 FROM sys.dual;  Capitals ----- The Soap</pre>
<pre>=&gt; SELECT LOWER('MR. SCOTT MCMILLAN') AS "Lowercase";  Lowercase ----- mr. scott mcmillan</pre>	<pre>SQL&gt; SELECT LOWER('MR. SCOTT MCMILLAN') "Lowercase" 2 FROM sys.dual;  Lowercase ----- mr. scott mcmillan</pre>
<pre>=&gt; SELECT LPAD('Page 1',15,'*.') AS "LPAD example";</pre>	<pre>SQL&gt; SELECT LPAD('Page 1',15,'*.') "LPAD example" 2 FROM sys.dual;</pre>

PostgreSQL 8.0.0	Oracle 10g
LPAD example ----- *.~.*~.*Page 1	LPAD example ----- *.~.*~.*Page 1
=> SELECT LTRIM('xyxXxyLAST WORD','xy') AS "LTRIM example";  LTRIM example ----- XxyLAST WORD	SQL> SELECT LTRIM('xyxXxyLAST WORD','xy') "LTRIM example" 2 FROM sys.dual; LTRIM example ----- XxyLAST WORD
=> SELECT REPLACE('JACK and JUE','J','BL') AS "Changes";  Changes ----- BLACK and BLUE	SQL> SELECT REPLACE('JACK and JUE','J','BL') "Changes" 2 FROM sys.dual; Changes ----- BLACK and BLUE
=> SELECT RPAD('MORRISON',12,'ab') AS "RPAD example";  RPAD example ----- MORRISONabab	SQL> SELECT RPAD('MORRISON',12,'ab') "RPAD example" 2 FROM sys.dual; RPAD example ----- MORRISONabab
=> SELECT RTRIM('BROWNINGyxXxyyx','xy') AS "RTRIM example";  RTRIM example ----- BROWNINGyxX	SQL> SELECT RTRIM('BROWNINGyxXxyyx','xy') "RTRIM example" 2 FROM sys.dual; RTRIM examp ----- BROWNINGyxX
=> SELECT SUBSTR('ABCDEFGF',3,4) AS "Substring";  Substring ----- CDEF	SQL> SELECT SUBSTR('ABCDEFGF',3,4) "Substring" 2 FROM sys.dual; Substring ----- CDEF
=> SELECT TRANSLATE('2KRW229', (> '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ', (> '9999999999XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX') AS "License";  License ----- 9XXX999	SQL> SELECT TRANSLATE('2KRW229', 2 '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ', 3 '9999999999XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX') "License" 4 FROM sys.dual; License ----- 9XXX999
=> SELECT TRANSLATE('2KRW229', (> '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ', '0123456789') -> AS "Translate example";  Translate example	SQL> SELECT TRANSLATE('2KRW229', 2 '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ', '0123456789') 3 "Translate example" 4 FROM sys.dual; Translate example



PostgreSQL 8.0.0	Oracle 10g
----- 2229	----- 2229
=> SELECT TRIM (0 FROM 0009872348900) AS "TRIM Example";  TRIM Example ----- 98723489	SQL> SELECT TRIM (0 FROM 0009872348900) "TRIM Example" 2 FROM sys.dual; TRIM Example ----- 98723489
=> SELECT UPPER('Large') AS "Uppercase";  Uppercase ----- LARGE	SQL> SELECT UPPER('Large') "Uppercase" 2 FROM sys.dual; Upper ----- LARGE

## 9.5. Funções e operadores para cadeias binárias

Esta seção descreve as funções e operadores disponíveis para examinar e manipular valores do tipo `bytea`.

O SQL define algumas funções para cadeias binárias com uma sintaxe especial, onde certas palavras chave, em vez de vírgulas, são utilizadas para separar os argumentos. Os detalhes estão na Tabela 9-10. Algumas funções também são implementadas utilizando a sintaxe regular de chamada de função (Consulte a Tabela 9-11).

**Tabela 9-10. Funções e operadores SQL para cadeias binárias**

Função	Tipo retornado	Descrição	Exemplo	Resultado
<code>cadeia_binária    cadeia_binária</code>	<code>bytea</code>	Concatenação de cadeias binárias	<code>'\\\\Post':: bytea    '\\\\047gres\\\\000' ::bytea</code>	<code>\\\\Post'gres\\\\000</code>
<code>octet_length( cadeia_binária)</code>	<code>integer</code>	Número de bytes da cadeia binária	<code>octet_length( 'Jo\\\\000sé':: bytea)</code>	5
<code>position( subcadeia in cadeia_binária)</code>	<code>integer</code>	Posição da subcadeia especificada	<code>position( '\\\\000om'::bytea in 'Th\\\\000omas':: bytea)</code>	3
<code>substring( cadeia_binária [from integer] [for integer])</code>	<code>bytea</code>	Extraí uma parte da cadeia binária	<code>substring( 'Th\\\\000omas':: bytea from 2 for 3)</code>	<code>h\\\\000o</code>
<code>trim([both] bytes from cadeia_binária)</code>	<code>bytea</code>	Remove a cadeia mais longa contendo apenas os bytes em bytes do início e do fim da cadeia binária.	<code>trim( '\\\\000'::bytea from '\\\\000Tom\\\\000' ::bytea)</code>	Tom
<code>get_byte( cadeia_binária,</code>	<code>integer</code>	Extraí byte da cadeia binária.	<code>get_byte( 'Th\\\\000omas'::</code>	109

Função	Tipo retornado	Descrição	Exemplo	Resultado
<code>deslocamento)</code>			<code>bytea, 4)</code>	
<code>set_byte(cadeia_binária, deslocamento, novo_valor)</code>	bytea	Define byte na cadeia binária.	<code>set_byte('Th\\000omas'::bytea, 4, 64)</code>	Th\000o@as
<code>get_bit(cadeia_binária, deslocamento)</code>	integer	Extraí bit da cadeia binária.	<code>get_bit('Th\\000omas'::bytea, 45)</code>	1
<code>set_bit(cadeia_binária, deslocamento, novo_valor)</code>	bytea	Define bit na cadeia binária.	<code>set_bit('Th\\000omas'::bytea, 45, 0)</code>	Th\000omAs

Estão disponíveis outras funções para manipulação de cadeias binárias, conforme mostrado na Tabela 9-11. Algumas são utilizadas internamente para implementar as funções de cadeias binárias do padrão SQL mostradas na Tabela 9-10.

**Tabela 9-11. Outras funções para cadeias binárias**

Função	Tipo retornado	Descrição	Exemplo	Resultado
<code>btrim(cadeia_binária bytea, bytes bytea)</code>	bytea	Remove do início e do fim da cadeia_binária, a cadeia mais longa contendo apenas os bytes em bytes	<code>btrim('\\000trim\\000'::bytea, '\\000'::bytea)</code>	trim
<code>length(cadeia_binária)</code>	integer	Comprimento da cadeia binária	<code>length('Jo\\000sê'::bytea)</code>	5
<code>decode(cadeia_binária text, tipo text)</code>	bytea	Decodifica a cadeia_binária previamente codificada com <code>encode()</code> . O tipo do parâmetro é o mesmo do <code>encode()</code> .	<code>decode('123\\000456', 'escape')</code>	123\000456
<code>encode(cadeia_binária bytea, tipo text)</code>	text	Codifica a cadeia_binária na representação somente ASCII. Os tipos suportados são: base64, hex, escape.	<code>encode('123\\000456'::bytea, 'escape')</code>	123\000456

## 9.6. Funções e operadores para cadeias de bits

Esta seção descreve as funções e operadores disponíveis para examinar e manipular cadeias de bits, ou seja, valores dos tipos `bit` e `bit varying`. Além dos operadores usuais de comparação, podem ser utilizados os operadores mostrados na Tabela 9-12. Devem ser do mesmo comprimento as cadeias de bits operandos de `&`, `|` e `#`. Ao ser feito o deslocamento de bits é preservado o comprimento original da cadeia de bits, conforme mostrado nos exemplos.

Tabela 9-12. Operadores para cadeias de bits

Operador	Descrição	Exemplo	Resultado
	concatenação	B'10001'    B'011'	10001011
&	AND bit a bit	B'10001' & B'01101'	00001
	OR bit a bit	B'10001'   B'01101'	11101
#	XOR bit a bit	B'10001' # B'01101'	11100
~	NOT bit a bit	~ B'10001'	01110
<<	deslocamento à esquerda bit a bit	B'10001' << 3	01000
>>	deslocamento à direita bit a bit	B'10001' >> 2	00100

As seguintes funções do padrão SQL funcionam em cadeias de bits, assim como em cadeias de caracteres: `length`, `bit_length`, `octet_length`, `position`, `substring`.

Além disso, é possível converter valores inteiros de e para o tipo `bit`. Alguns exemplos:

```
44::bit(10)           0000101100
44::bit(3)            100
cast(-44 as bit(12))  111111010100
'1110'::bit(4)::integer 14
```

Deve ser observado que converter para apenas um “bit” significa converter para `bit(1)` e, portanto, o resultado será apenas o bit menos significativo do inteiro.

**Nota:** Antes do PostgreSQL 8.0, a conversão de um inteiro para `bit(n)` copiava os *n* bits mais à esquerda do inteiro, enquanto agora copia os *n* bits mais à direita. Também, converter um inteiro para uma cadeia de bits mais larga que o próprio inteiro, estende o sinal para à esquerda.

## 9.7. Correspondência com padrão

O PostgreSQL disponibiliza três abordagens distintas para correspondência com padrão: o operador `LIKE` tradicional do SQL; o operador mais recente `SIMILAR TO` (adicionado ao SQL:1999); e as expressões regulares no estilo POSIX. Além disso, também está disponível a função de correspondência com padrão `substring`, que utiliza expressões regulares tanto no estilo `SIMILAR TO` quanto no estilo POSIX.

**Dica:** Havendo necessidade de correspondência com padrão acima destas, deve ser considerado o desenvolvimento de uma função definida pelo usuário em Perl ou Tcl.

### 9.7.1. LIKE

```
cadeia_de_caracteres LIKE padrão [ESCAPE caractere_de_escape]
cadeia_de_caracteres NOT LIKE padrão [ESCAPE caractere_de_escape]
```

Cada *padrão* define um conjunto de cadeias de caracteres. A expressão `LIKE` retorna verdade se a *cadeia\_de\_caracteres* estiver contida no conjunto de cadeias de caracteres representado pelo *padrão*; como esperado, a expressão `NOT LIKE` retorna falso quando `LIKE` retorna verdade, e vice-versa, e a expressão equivalente é `NOT (cadeia_de_caracteres LIKE padrão)`.

Quando o *padrão* não contém os caracteres percentagem ou sublinhado, o *padrão* representa apenas a própria cadeia de caracteres; neste caso `LIKE` atua como o operador igual. No *padrão* o caractere sublinhado (`_`) representa (corresponde a) qualquer um único caractere; o caractere percentagem (`%`) corresponde a qualquer cadeia com zero ou mais caracteres.

Alguns exemplos:

```
'abc' LIKE 'abc'      verdade
'abc' LIKE 'a%'       verdade
'abc' LIKE '_b_'      verdade
```

```
'abc' LIKE 'c'      falso
```

A correspondência com padrão `LIKE` sempre abrange toda a cadeia de caracteres. Para haver correspondência com o padrão em qualquer posição da cadeia de caracteres, o padrão deve começar e terminar pelo caractere percentagem.

Para corresponder ao próprio caractere sublinhado ou percentagem, sem corresponder a outros caracteres, estes caracteres devem ser precedidos pelo caractere de escape no padrão. O caractere de escape padrão é a contrabarra, mas pode ser definido um outro caractere através da cláusula `ESCAPE`. Para corresponder ao próprio caractere de escape, devem ser escritos dois caracteres de escape.

Deve ser observado que a contrabarra também possui significado especial nos literais cadeias de caracteres e, portanto, para escrever em uma constante um padrão contendo uma contrabarra devem ser escritas duas contrabarras no comando SQL. Assim sendo, para escrever um padrão que corresponda ao literal contrabarra é necessário escrever quatro contrabarras no comando, o que pode ser evitado escolhendo um caractere de escape diferente na cláusula `ESCAPE`; assim a contrabarra deixa de ser um caractere especial para o `LIKE` (Mas continua sendo especial para o analisador de literais cadeias de caracteres e, por isso, continuam sendo necessárias duas contrabarras).

Também é possível fazer com que nenhum caractere sirva de escape declarando `ESCAPE ''`. Esta declaração tem como efeito desabilitar o mecanismo de escape, tornando impossível anular o significado especial dos caracteres sublinhado e percentagem no padrão.

Alguns exemplos: (N. do T.)

```
-- Neste exemplo a contrabarra única é consumida pelo analisador de literais
-- cadeias de caracteres, e não anula o significado especial do _

=> SELECT tablename FROM pg_tables WHERE tablename LIKE '%g\_o%' ESCAPE '\\';

  tablename
  -----
 pg_group
 pg_operator
 pg_opclass
 pg_largeobject
(4 linhas)

-- Neste exemplo somente uma das duas contrabarras é consumida pelo analisador
-- de literais cadeias de caracteres e, portanto, a segunda contrabarra anula
-- o significado especial do _

=> SELECT tablename FROM pg_tables WHERE tablename LIKE '%g\\\_o%' ESCAPE '\\';

  tablename
  -----
 pg_operator
 pg_opclass
(2 linhas)

-- No Oracle não são necessárias duas contrabarras, como mostrado abaixo:

SQL> SELECT view_name FROM all_views WHERE view_name LIKE 'ALL\\_%' ESCAPE '\\';
```

Pode ser utilizada a palavra chave `ILIKE` no lugar de `LIKE` para fazer a correspondência não diferenciar letras maiúsculas de minúsculas, conforme o idioma ativo.<sup>3</sup> Isto não faz parte do padrão SQL, sendo uma extensão do PostgreSQL.

O operador `~~` equivale ao `LIKE`, enquanto `~~*` corresponde ao `ILIKE`. Também existem os operadores `!~~` e `!~~*`, representando o `NOT LIKE` e o `NOT ILIKE` respectivamente. Todos estes operadores são específicos do PostgreSQL.

### 9.7.2. Expressões regulares do SIMILAR TO

```
cadeia_de_caracteres SIMILAR TO padrão [ESCAPE caractere_de_escape]
cadeia_de_caracteres NOT SIMILAR TO padrão [ESCAPE caractere_de_escape]
```

O operador `SIMILAR TO` retorna verdade ou falso conforme o padrão corresponda ou não à cadeia de caracteres fornecida. Este operador é muito semelhante ao `LIKE`, exceto por interpretar o padrão utilizando a definição de expressão regular do padrão SQL. As expressões regulares do padrão SQL são um cruzamento curioso entre a notação do `LIKE` e a notação habitual das expressões regulares.

Da mesma forma que o `LIKE`, o operador `SIMILAR TO` somente é bem-sucedido quando o padrão corresponde a toda cadeia de caracteres; é diferente do praticado habitualmente nas expressões regulares, onde o padrão pode corresponder a qualquer parte da cadeia de caracteres. Também como o `LIKE`, o operador `SIMILAR TO` utiliza `_` e `%` como caracteres curinga, representando qualquer um único caractere e qualquer cadeia de caracteres, respectivamente (são comparáveis ao `.` e ao `*` das expressões regulares POSIX).

Além destas funcionalidades pegadas emprestada do `LIKE`, o `SIMILAR TO` suporta os seguintes metacaracteres para correspondência com padrão pegos emprestado das expressões regulares POSIX:

- `|` representa alternância (uma das duas alternativas).
- `*` representa a repetição do item anterior zero ou mais vezes.
- `+` representa a repetição do item anterior uma ou mais vezes.
- Os parênteses `()` podem ser utilizados para agrupar itens em um único item lógico.
- A expressão de colchetes `[...]` especifica uma classe de caracteres, do mesmo modo que na expressão regular POSIX.

Deve ser observado que as repetições limitadas (`?` e `{...}`) não estão disponíveis, embora existam no POSIX. Além disso, o ponto `.` não é um metacaractere.

Da mesma forma que no `LIKE`, a contrabarra desabilita o significado especial de qualquer um dos metacaracteres; ou pode ser especificado um caractere de escape diferente por meio da cláusula `ESCAPE`.

Alguns exemplos:

```
'abc' SIMILAR TO 'abc'      verdade
'abc' SIMILAR TO 'a'        falso
'abc' SIMILAR TO '%(b|d)%'  verdade
'abc' SIMILAR TO '(b|c)%'   falso
```

A função `substring` com três parâmetros, `substring(cadeia_de_caracteres FROM padrão FOR caractere_de_escape)`, permite extrair a parte da cadeia de caracteres que corresponde ao padrão da expressão regular SQL:1999. Assim como em `SIMILAR TO`, o padrão especificado deve corresponder a toda a cadeia de caracteres, senão a função falha e retorna nulo. Para indicar a parte do padrão que deve ser retornada em caso de sucesso, o padrão deve conter duas ocorrências do caractere de escape seguidas por aspas (`"`). É retornado o texto correspondente à parte do padrão entre estas marcas.

Alguns exemplos:

```
substring('foobar' FROM '%"o_b#"' FOR '#')    oob
substring('foobar' FROM '%"o_b#"' FOR '#')    NULL
```

### 9.7.3. Expressões regulares POSIX

A Tabela 9-13 mostra os operadores disponíveis para correspondência com padrão utilizando as expressões regulares POSIX.

**Tabela 9-13. Operadores de correspondência para expressões regulares**

Operador	Descrição	Exemplo
<code>~</code>	Corresponde à expressão regular, diferenciando maiúsculas e minúsculas	<code>'thomas' ~</code> <code>'.*thomas.*'</code>
<code>~*</code>	Corresponde à expressão regular, não diferenciando maiúsculas e minúsculas	<code>'thomas' ~*</code> <code>'.*Thomas.*'</code>
<code>!~</code>	Não corresponde à expressão regular, diferenciando maiúsculas e minúsculas	<code>'thomas' !~</code> <code>'.*Thomas.*'</code>

Operador	Descrição	Exemplo
<code>!~*</code>	Não corresponde à expressão regular, não diferenciando maiúsculas e minúsculas	<code>'thomas' !~*</code> <code>'.*vadim.*'</code>

As expressões regulares POSIX fornecem uma forma mais poderosa para correspondência com padrão que os operadores `LIKE` e `SIMILAR TO`. Muitas ferramentas do Unix, como `egrep`, `sed` e `awk`, utilizam uma linguagem para correspondência com padrão semelhante à descrita aqui.

Uma expressão regular é uma sequência de caracteres contendo uma definição abreviada de um conjunto de cadeias de caracteres (um *conjunto regular*). Uma cadeia de caracteres é dita correspondendo a uma expressão regular se for membro do conjunto regular descrito pela expressão regular. Assim como no `LIKE`, os caracteres do padrão correspondem exatamente aos caracteres da cadeia de caracteres, a não ser quando forem caracteres especiais da linguagem da expressão regular — porém, as expressões regulares utilizam caracteres especiais diferentes dos utilizados pelo `LIKE`. Diferentemente dos padrões do `LIKE`, uma expressão regular pode corresponder a qualquer parte da cadeia de caracteres, a não ser que a expressão regular seja explicitamente ancorada ao início ou ao final da cadeia de caracteres.

Alguns exemplos:

```
'abc' ~ 'abc'      verdade
'abc' ~ '^a'       verdade
'abc' ~ '(b|d)'    verdade
'abc' ~ '^ (b|c)'  falso
```

A função `substring` com dois parâmetros, `substring(cadeia_de_caracteres FROM padrão)`, permite extrair a parte da cadeia de caracteres que corresponde ao padrão da expressão regular POSIX. A função retorna nulo quando não há correspondência, senão retorna a parte do texto que corresponde ao padrão. Entretanto, quando o padrão contém parênteses, é retornada a parte do texto correspondendo à primeira subexpressão entre parênteses (aquela cujo abre parênteses vem primeiro). Podem ser colocados parênteses envolvendo toda a expressão, se for desejado utilizar parênteses em seu interior sem disparar esta exceção. Se for necessária a presença de parênteses no padrão antes da subexpressão a ser extraída, veja os parênteses não-capturantes descritos abaixo.

Alguns exemplos:

```
substring('foobar' from 'o.b')    oob
substring('foobar' from 'o(.)b')  o
```

As expressões regulares do PostgreSQL são implementadas utilizando um pacote escrito por Henry Spencer (<http://www.hq.nasa.gov/office/pao/History/alsj/henry.html>). Grande parte da descrição das expressões regulares abaixo foi copiada textualmente desta parte de seu manual.

### 9.7.3.1. Detalhes das expressões regulares

As expressões regulares (ERs), conforme definidas no POSIX 1003.2, estão presentes em duas formas: ERs *estendidas* ou EREs (aproximadamente as do `egrep`), e ERs *básicas* ou ERBs (aproximadamente as do `ed`). O PostgreSQL suporta as duas formas, e também implementa algumas extensões que não fazem parte do padrão POSIX, mas que são muito utilizadas por estarem disponíveis em linguagens de programação como Perl e Tcl. As ERs que utilizam as extensões não-POSIX são chamadas de ERs *avançadas* ou ERAs nesta documentação. As ERAs são quase um superconjunto exato das EREs, mas as ERBs possuem várias notações incompatíveis (bem como são muito mais limitadas). Primeiro são descritas as formas ERA e ERE, indicando as funcionalidades que se aplicam somente as ERAs, e depois descrevendo como as ERBs diferem.

**Nota:** A forma das expressões regulares aceitas pelo PostgreSQL pode ser escolhida definindo o parâmetro em tempo de execução `regex_flavor`. A definição usual é `advanced`, mas pode ser escolhido `extended` para o máximo de compatibilidade com as versões do PostgreSQL anteriores a 7.4.

Uma expressão regular é definida como uma ou mais *ramificações* separadas por `|`. Corresponde a tudo que corresponda a uma de suas ramificações.

Uma ramificação é formada por zero ou mais *átomos quantificados*, ou *restrições*, concatenados. Corresponde à correspondência para a primeira, seguida pela correspondência para a segunda, etc.; uma ramificação vazia corresponde à cadeia de caracteres vazia.

Um átomo quantificado é um *átomo* possivelmente seguido por um único *quantificador*. Sem o quantificador, corresponde à correspondência para o átomo. Com o quantificador, pode corresponder a um número de ocorrências do átomo. Um *átomo* pode ser uma das possibilidades mostradas na Tabela 9-14. Os quantificadores possíveis e seus significados estão mostrados na Tabela 9-15.

Uma *restrição* corresponde a uma cadeia de caracteres vazia, mas corresponde apenas quando determinadas condições são satisfeitas. Uma restrição pode ser utilizada onde um átomo pode ser utilizado, exceto que não pode ser seguida por um quantificador. As restrições simples estão mostradas na Tabela 9-16; algumas outras restrições são descritas posteriormente.

**Tabela 9-14. Átomos de expressões regulares**

Átomo	Descrição
$(er)$	(onde $er$ é qualquer expressão regular) corresponde à correspondência para $er$ , com a correspondência marcada para um possível relato
$(?:er)$	o mesmo acima, mas a correspondência não está marcada para relato (um conjunto de parênteses “não-capturante”) (somente ERAs)
$.$	corresponde a qualquer um único caractere
$[caracteres]$	uma <i>expressão de colchetes</i> , correspondendo a qualquer um dos <i>caracteres</i> (consulte a Seção 9.7.3.2 para obter mais detalhes)
$\backslash k$	(onde $k$ é um caractere não alfanumérico) corresponde a este caractere tomado como um caractere ordinário, por exemplo, $\backslash \backslash$ corresponde ao caractere contrabarra
$\backslash c$	onde $c$ é alfanumérico (possivelmente seguido por outros caracteres) é um <i>escape</i> , consulte a Seção 9.7.3.3 (ERAs somente; nas EREs e nas ERBs corresponde ao $c$ )
$\{$	quando seguido por um caractere que não seja um dígito, corresponde ao caractere abre chaves $\{$ ; quando seguido por um dígito, é o início de um <i>limite</i> (veja abaixo)
$x$	onde $x$ é um único caractere sem nenhum outro significado, corresponde a este caractere

Uma expressão regular não pode terminar por  $\backslash$ .

**Nota:** Lembre-se que a contrabarra ( $\backslash$ ) tem um significado especial nos literais cadeias de caracteres do PostgreSQL. Para escrever em uma constante um padrão contendo uma contrabarra, devem ser escritas duas contrabarras na declaração.

**Tabela 9-15. Quantificadores de expressão regular**

Quantificador	Corresponde
$*$	uma sequência de 0 ou mais correspondências do átomo
$+$	uma sequência de 1 ou mais correspondências do átomo
$?$	uma sequência de 0 ou 1 correspondência do átomo
$\{m\}$	uma sequência de exatamente $m$ correspondências do átomo
$\{m, \}$	uma sequência de $m$ ou mais correspondências do átomo
$\{m, n\}$	uma sequência de $m$ a $n$ (inclusive) correspondências do átomo; $m$ não pode ser maior do que $n$
$*?$	versão não-voraz do $*$
$+?$	versão não-voraz do $+$
$??$	versão não-voraz do $?$
$\{m\}?$	versão não-voraz do $\{m\}$

Quantificador	Corresponde
$\{m, \}$ ?	versão não-voraz do $\{m, \}$
$\{m, n\}$ ?	versão não-voraz do $\{m, n\}$

As formas que utilizam  $\{ \dots \}$  são conhecidas como *limites*. Os números  $m$  e  $n$  dentro dos limites são inteiros decimais sem sinal, com valores permitidos entre 0 e 255, inclusive.

Quantificadores *não-vorazes* (disponíveis apenas nas ERAs) correspondem às mesmas possibilidades que seus semelhantes normais (*vorazes*), mas preferem o menor número em vez do maior número de correspondências. Consulte a Seção 9.7.3.5 para obter mais detalhes.

**Nota:** Um quantificador não pode vir imediatamente após outro quantificador. Um quantificador não pode começar uma expressão ou uma subexpressão, ou seguir  $\wedge$  ou  $|$ .

**Tabela 9-16. Restrições de expressão regular**

Restrição	Descrição
$\wedge$	corresponde no início da cadeia de caracteres
$\$$	corresponde no fim da cadeia de caracteres
$(?=er)$	<i>olhar à frente positiva</i> <sup>a</sup> — corresponde em qualquer ponto a partir de onde começa a parte da cadeia de caracteres que corresponde à $er$ (somente ERAs)
$(?!er)$	<i>olhar à frente negativa</i> <sup>b</sup> — corresponde em qualquer ponto a partir de onde começa uma parte da cadeia de caracteres que não corresponde à $er$ (somente ERAs)

Notas:

- $(?=)$ padrão) — olhar à frente positiva (*positive lookahead*) corresponde à cadeia de caracteres de procura em qualquer ponto onde começa uma cadeia de caracteres correspondendo ao padrão. Esta é uma correspondência não capturante, ou seja, a correspondência não é capturada para um possível uso posterior. Por exemplo `'Windows (=?95|98|NT|2000)'` corresponde a “Windows” em “Windows 2000”, mas não a “Windows” em “Windows 3.1”. Olhar à frente não consome caracteres, ou seja, após ocorrer a correspondência a procura pela próxima ocorrência começa imediatamente após a última ocorrência, e não após os caracteres compreendidos pelo olhar à frente. Introduction to Regular Expressions (<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/js56jsgrpregexpsyntax.asp>) (N. do T.)
- $(?!)$ padrão) — olhar à frente negativa (*negative lookahead*) corresponde à cadeia de caracteres de procura em qualquer ponto onde começa uma cadeia de caracteres que não corresponde ao padrão. Esta é uma correspondência não capturante, ou seja, a correspondência não é capturada para um possível uso posterior. Por exemplo `'Windows (?!95|98|NT|2000)'` corresponde a “Windows” em “Windows 3.1”, mas não a “Windows” em “Windows 2000”. Olhar à frente não consome caracteres, ou seja, após ocorrer a correspondência a procura pela próxima ocorrência começa imediatamente após a última ocorrência, e não após os caracteres compreendidos pelo olhar à frente. Introduction to Regular Expressions (<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/js56jsgrpregexpsyntax.asp>) (N. do T.)

Alguns exemplos: (N. do T.)

```
substring('Windows 2000' FROM 'Windows (=?95|98|NT|2000)')  Windows
substring('Windows 3.1' FROM 'Windows (=?95|98|NT|2000)')  NULL
substring('Windows 2000' FROM 'Windows (?!95|98|NT|2000)')  NULL
substring('Windows 3.1' FROM 'Windows (?!95|98|NT|2000)')  Windows
```

As restrições de olhar à frente não podem conter *referências para trás* (consulte a Seção 9.7.3.3), e todos os parênteses dentro das mesmas são considerados não-capturantes.

### 9.7.3.2. Expressões de colchetes

Uma *expressão de colchetes* é uma lista de caracteres entre  $[ ]$ . Normalmente corresponde a qualquer um único caractere da lista (mas veja abaixo). Se a lista começar por  $\wedge$ , corresponde a qualquer um único caractere não presente no restante da



lista (mas veja abaixo). Se dois caracteres da lista estiverem separados por `-`, isto representa a forma abreviada de todos os caracteres entre estes dois (inclusive) na sequência de classificação (*collating sequence*<sup>4</sup>) como, por exemplo, `[0-9]` em ASCII corresponde a qualquer dígito decimal. É ilegal dois intervalos compartilharem uma mesma extremidade como, por exemplo, `a-c-e`. Os intervalos são dependentes da sequência de classificação dos caracteres, e os programas portáteis devem evitar esta dependência.

Para incluir o literal `]` na lista, deve ser feito com que seja o primeiro caractere (seguindo um possível `^`). Para incluir o literal `-`, deve ser feito com que seja o primeiro ou o último caractere, ou a segunda extremidade de um intervalo. Para utilizar o literal `-` como a primeira extremidade do intervalo, deve-se colocá-lo entre `[ . e . ]` para torná-lo um elemento de classificação (veja abaixo). Com exceção destes caracteres, algumas combinações utilizando `[` (veja os próximos parágrafos), e escapes (somente ERAs), todos os outros caracteres especiais perdem seu significado especial dentro da expressão de colchetes. Em particular, `\` não é especial ao seguir as regras das EREs ou ERBs, embora seja especial (como introdução de escape) nas ERAs.

Dentro da expressão de colchetes, o elemento de classificação (um caractere, uma sequência de vários caracteres que é classificada como sendo um único caractere, ou o nome de uma sequência de classificação) entre `[ . e . ]` representa a sequência de caracteres deste elemento de classificação. A sequência é um único elemento da lista da expressão de colchetes. Uma expressão de colchetes contendo um elemento de classificação com vários caracteres pode, portanto, corresponder a mais de um caractere. Por exemplo, se a sequência de classificação incluir o elemento de classificação `ch`, então a expressão regular `[ [ . ch . ] ] * c` corresponde aos cinco primeiros caracteres de `chchccc`.

**Nota:** Atualmente o PostgreSQL não possui elementos de classificação de vários caracteres. Esta informação descreve um possível comportamento futuro.

Dentro da expressão de colchetes, um elemento de classificação entre `[ = e = ]` é uma classe de equivalência, representando as sequências de caracteres de todos os elementos de classificação equivalentes a este elemento, incluindo o próprio (Se não existirem outros elementos de classificação equivalentes, o tratamento é como se os delimitadores envoltórios fossem `[ . e . ]`). Por exemplo, se `o` e `^` são membros de uma classe de equivalência, então `[ [=o= ] ]`, `[ [=^= ] ]` e `[ o^ ]` são todos sinônimos. Uma classe de equivalência não pode ser a extremidade de um intervalo.

Dentro da expressão de colchetes, o nome de uma classe de caracteres entre `[ : e : ]` representa a lista de todos os caracteres pertencentes a esta classe. Os nomes das classes de caracteres padrão são: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`. Estes nomes representam as classes de caracteres definidas em `ctype`. O idioma pode fornecer outras. Uma classe de caracteres não pode ser usada como extremidade de um intervalo.

Existem dois casos especiais de expressões de colchetes: as expressões de colchetes `[[:<:]]` e `[[:>:]]` são restrições, correspondendo a cadeias de caracteres vazias no início e no fim da palavra, respectivamente. Uma *palavra é definida como a sequência de caracteres de palavra, que não é nem precedida nem seguida por caractere de palavra*. Um caractere de palavra é um caractere `alnum` (conforme definido em `ctype`) ou um sublinhado. Isto é uma extensão, compatível mas não especificada pelo POSIX 1003.2, devendo ser utilizada com cautela em programas onde se deseja a portabilidade para outros sistemas. Os escapes de restrição descritos abaixo geralmente são preferíveis (não são mais padrão que estes, mas com certeza são mais fáceis de serem digitados).

### 9.7.3.3. Escapes de expressão regular

*Escapes* são sequências especiais começando por `\` seguida por um caractere alfanumérico. Os escapes se apresentam em diversas variedades: entrada de caractere, abreviaturas de classe, escapes de restrição e referências para trás. Uma `\` seguida por um caractere alfanumérico que não constitua um escape válido é ilegal nas ERAs. Nas EREs não existem escapes: fora da expressão de colchetes, uma `\` seguida por um caractere alfanumérico representa tão somente este caractere como um caractere comum, e dentro da expressão de colchetes, a `\` é um caractere comum; Esta última é uma verdadeira incompatibilidade entre EREs e ERAs.

Os *escapes de entrada de caractere* existem para tornar mais fácil a especificação de caracteres não-imprimíveis, ou de alguma outra maneira inconvenientes, nas expressões regulares. Estão mostrados na Tabela 9-17.

Os *escapes de abreviatura de classe* fornecem abreviaturas para certas classes de caracteres comumente utilizadas. Estão mostrados na Tabela 9-18.

O *escape de restrição* é uma restrição, correspondendo à cadeia de caracteres vazia caso ocorram determinadas condições, escrita como um escape. Estão mostrados na Tabela 9-19.

Uma *referência para trás* ( $\backslash n$ ) corresponde à mesma cadeia de caracteres correspondida pela subexpressão entre parênteses anterior especificada pelo número  $n$  (consulte a Tabela 9-20). Por exemplo,  $([bc])\backslash 1$  corresponde a  $bb$  ou  $cc$ , mas não a  $bc$  ou  $cb$ . A subexpressão deve preceder inteiramente a referência para trás na expressão regular. As subexpressões são numeradas na ordem de seu parêntese de abertura. Parênteses não-capturantes não definem subexpressões.

**Nota:** Deve-se ter em mente que a  $\backslash$  de abertura do escape deve ser dobrada ao se entrar o padrão usando uma constante cadeia de caracteres do SQL. Por exemplo:

```
'123' ~ '^\\d{3}' verdade
```

**Tabela 9-17. Escapes entrada de caractere em expressão regular**

Escape	Descrição
$\backslash a$	caractere de alerta ( <code>bell</code> ), como na linguagem C
$\backslash b$	voltar apagando ( <code>backspace</code> ), como na linguagem C
$\backslash B$	sinônimo de $\backslash$ para ajudar a reduzir a necessidade de contrabarras dobradas
$\backslash cX$	(onde $X$ é qualquer caractere) o caractere cujos 5 bits de mais baixa ordem são os mesmos de $X$ , e cujos outros bits são todos zero
$\backslash e$	o caractere cujo nome da sequência de classificação é <code>ESC</code> ou, na falta deste, o caractere com valor octal 033
$\backslash f$	avanço de formulário, como na linguagem C
$\backslash n$	nova-linha, como na linguagem C
$\backslash r$	retorno-de-carro, como na linguagem C
$\backslash t$	tabulação horizontal, como na linguagem C
$\backslash uwxyz$	(onde $wxyz$ são exatamente quatro dígitos hexadecimais) o caractere Unicode $U+wxyz$ na ordem de bytes local
$\backslash Ustuvwxyz$	(onde $stuvwxyz$ são exatamente oito dígitos hexadecimais) reservado para uma extensão hipotética do Unicode para 32 bits
$\backslash v$	tabulação vertical, como na linguagem C
$\backslash xhhh$	(onde $hhh$ é qualquer sequência de dígitos hexadecimais) o caractere cujo valor hexadecimal é $0xhhh$ (um único caractere não importando quantos dígitos hexadecimais são utilizados)
$\backslash 0$	o caractere cujo valor é 0
$\backslash xy$	(onde $xy$ são exatamente dois dígitos octais, e não é uma <i>referência para trás</i> ) o caractere cujo valor octal é $0xy$
$\backslash xyz$	(onde $xyz$ são exatamente três dígitos octais, e não é uma <i>referência para trás</i> ) o caractere cujo valor octal é $0xyz$

Os dígitos hexadecimais são 0-9, a-f e A-F. Os dígitos octais são 0-7.

Os escapes de entrada de caractere são sempre tomados como caracteres comuns. Por exemplo,  $\backslash 135$  é `] em ASCII`, mas  $\backslash 135$  não termina uma expressão de colchetes.

**Tabela 9-18. Escapes de abreviatura de classe em expressão regular**

Escape	Descrição
\d	[[ :digit: ]]
\s	[[ :space: ]]
\w	[[ :alnum: ]_] (deve ser observado que inclui o sublinhado)
\D	[^ :digit: ]]
\S	[^ :space: ]]
\W	[^ :alnum: ]_] (deve ser observado que inclui o sublinhado)

Dentro das expressões de colchetes, \d, \s e \w perdem seus colchetes externos, e \D, \S e \W são ilegais (Portanto, por exemplo, [a-c\d] equivale a [a-c[:digit:]]). Também, [a-c\D], que equivale a [a-c[^:digit:]], é ilegal).

**Tabela 9-19. Escapes de restrição em expressão regular**

Escape	Descrição
\A	corresponde apenas no início da cadeia de caracteres (veja na Seção 9.7.3.5 como isto difere de ^)
\m	corresponde apenas no início da palavra
\M	corresponde apenas no final da palavra
\Y	corresponde apenas no início ou no final da palavra
\Y	corresponde apenas no ponto que não é nem o início nem o final da palavra
\Z	corresponde apenas no final da cadeia de caracteres (veja na Seção 9.7.3.5 como isto difere de \$)

Uma palavra é definida como na especificação de [[ :<: ]] e [[ :>: ]] acima. Os escapes de restrição são ilegais dentro de expressões de colchetes.

**Tabela 9-20. Referências para trás em expressões regulares**

Escape	Descrição
\m	(onde <i>m</i> é um dígito diferente de zero) uma referência para trás à <i>m</i> 'ésima subexpressão
\mnn	(onde <i>m</i> é um dígito diferente de zero, e <i>nn</i> são mais alguns dígitos, e o valor decimal <i>mnn</i> não é maior do que o número de parênteses capturantes fechados até este ponto) uma referência para trás à <i>mnn</i> 'ésima subexpressão

**Nota:** Existe uma ambigüidade histórica inerente entre as entradas de caractere em octal para escapes e referências para trás, que é solucionado por heurística, como mostrado acima. Um zero na frente sempre indica um escape octal. Um único dígito diferente de zero, não seguido por outro dígito, é sempre assumido como uma referência para trás. Uma sequência de vários dígitos não começada por zero é assumida como uma referência para trás se vier após uma subexpressão adequada (ou seja, o número está na faixa legal de referências para trás), senão é assumida como um octal.

#### 9.7.3.4. Metassintaxe em expressão regular

Além da sintaxe principal descrita acima, existem algumas formas especiais e uma miscelânea de facilidades de sintaxe disponíveis.

Normalmente, a variedade da expressão regular sendo utilizada é determinada por `regex_flavor`, mas pode ser mudada pelo prefixo *diretriz*. Se a expressão regular começar por `***:`, o restante da expressão regular é assumido como sendo uma ARE a despeito de `regex_flavor`. Se a expressão regular começar por `***=`, o restante da expressão regular é assumido como sendo um literal cadeia de caracteres, com todos os caracteres considerados como sendo caracteres comuns.

Uma ERA pode começar por *opções incorporadas*: uma sequência (`?xyz`) (onde `xyz` é um ou mais caracteres alfabéticos) especifica opções que afetam o restante da expressão regular. Estas opções substituem qualquer opção previamente determinada (incluindo a variedade da expressão regular e a diferenciação de maiúsculas e minúsculas). As letras das opções disponíveis estão mostradas na Tabela 9-21.

**Tabela 9-21. Letras de opção incorporada em ERA**

Opção	Descrição
b	o restante da expressão regular é uma ERB
c	correspondência fazendo distinção entre letras maiúsculas e minúsculas (substitui o tipo do operador)
e	o restante da expressão regular é uma ERE
i	correspondência não fazendo distinção entre letras maiúsculas e minúsculas (consulte a Seção 9.7.3.5) (substitui o tipo do operador)
m	sinônimo histórico para n
n	correspondência sensível à nova-linha (consulte a Seção 9.7.3.5)
p	correspondência sensível à nova-linha parcial (consulte a Seção 9.7.3.5)
q	o restante da expressão regular é um literal cadeia de caracteres (“entre aspas”), todos caracteres são comuns
s	correspondência não sensível à nova-linha (padrão)
t	sintaxe amarrada (padrão; veja abaixo)
w	correspondência sensível a nova-linha parcial inversa (“estranha”) (consulte a Seção 9.7.3.5)
x	sintaxe expandida (veja abaixo)

As opções incorporadas se tornam ativas no fecha parênteses “)” que termina a sequência. Podem aparecer apenas no início de uma ERA (após a diretriz `***:`, caso exista).

Além da sintaxe de expressão regular usual (*amarrada*), na qual todos os caracteres possuem significado, existe uma sintaxe *expandida*, disponível especificando-se a opção incorporada `x`. Na sintaxe expandida, os caracteres de espaço-em-branco na expressão regular são ignorados, assim como todos os caracteres entre a `#` e a próxima nova-linha (ou o fim da expressão regular). Isto permite colocar parágrafos e comentários em uma expressão regular complexa. Existem três exceções para esta regra básica:

- um caractere de espaço em branco ou `#` precedido por `\` é retido
- espaço em branco ou `#` dentro da expressão de colchetes é retido
- espaço em branco e comentários são ilegais dentro de símbolos multicaractere, como o `(?:`

Para esta finalidade os caracteres de espaço-em-branco são espaço, tabulação, nova-linha e qualquer outro caractere que pertença a classe de caracteres *space*.

Por fim, em uma ERA, fora das expressões de colchetes, a sequência `(?#ttt)` (onde `ttt` é qualquer texto não contendo um `)`) é um comentário, completamente ignorado. Novamente, isto não é permitido entre os caracteres dos símbolos multicaractere, como o `(?:`. Estes comentários são mais um artefato histórico do que uma funcionalidade útil, e sua utilização está obsoleta; deve ser usada a sintaxe expandida em seu lugar.

Nenhuma destas extensões da metassintaxe está disponível quando uma diretriz inicial `***=` especificar que a entrada do usuário deve ser tratada como um literal cadeia de caracteres em vez de uma expressão regular.

### 9.7.3.5. Regras de correspondência de expressão regular

Quando uma expressão regular corresponde a mais de uma parte de uma dada cadeia de caracteres, a expressão regular corresponde à parte que começa primeiro. Se a expressão regular puder corresponder a mais de uma parte da cadeia de caracteres começando neste ponto, então corresponderá à correspondência mais longa possível, ou corresponderá à correspondência mais curta possível, dependendo se a expressão regular é *voraz* (*greedy*) ou *não-voraz* (*non-greedy*).

Se a expressão regular é voraz ou não, é determinado pelas seguintes regras:

- A maioria dos átomos, e todas as restrições, não possuem atributos de voracidade (porque não podem corresponder a quantidades variáveis de texto de qualquer forma).
- A adição de parênteses em torno da expressão regular não muda sua voracidade.
- Um átomo quantificado por um quantificador de repetição fixo ( $\{m\}$  ou  $\{m\}?$ ) possui a mesma voracidade (possivelmente nenhuma) do próprio átomo.
- Um átomo quantificado por outros quantificadores normais (incluindo  $\{m, n\}$  com  $m$  igual a  $n$ ) é voraz (prefere a correspondência mais longa).
- Um átomo quantificado por um quantificador não voraz (incluindo  $\{m, n\}?$  com  $m$  igual a  $n$ ) é não-voraz (prefere a correspondência mais curta).
- Uma ramificação — ou seja, uma expressão regular que não possua o operador de nível mais alto  $|$  — possui a mesma voracidade do primeiro átomo quantificado nesta que possui o atributo de voracidade.
- Uma expressão regular formada por duas ou mais ramificações conectadas pelo operador  $|$  é sempre voraz.

As regras acima associam os atributos de voracidade não apenas aos átomos quantificados individualmente, mas também com as ramificações e as expressões regulares inteiras contendo átomos quantificados. Isto significa que a correspondência é feita de uma forma que a ramificação, ou toda a expressão regular, corresponda à parte da cadeia de caracteres mais longa ou mais curta possível *como um todo*. Uma vez que seja determinado o comprimento de toda a correspondência, a parte desta que corresponde a uma determinada subexpressão é determinada com base no atributo de voracidade desta subexpressão, com as subexpressões que começam primeiro na expressão regular tendo prioridade sobre as que começam depois.

Exemplo do que isto significa:

```
SELECT substring('XY1234Z', 'Y*([0-9]{1,3})');
Resultado: 123
SELECT substring('XY1234Z', 'Y*?([0-9]{1,3})');
Resultado: 1
```

No primeiro caso, a expressão regular como um todo é voraz porque  $Y^*$  é voraz. Pode corresponder a partir do  $Y$ , e corresponde à cadeia de caracteres mais longa possível começando neste ponto, ou seja,  $Y123$ . A saída é a parte entre parênteses da expressão, ou seja,  $123$ . No segundo caso, a expressão regular como um todo não é voraz porque  $Y^*?$  é não voraz. Pode corresponder a partir do  $Y$ , e corresponde à cadeia de caracteres mais curta possível começando neste ponto, ou seja,  $Y1$ . A subexpressão  $[0-9]{1,3}$  é voraz, mas não pode mudar a decisão para o comprimento de correspondência geral; portanto é forçada a corresponder apenas ao  $1$ .

Em resumo, quando a expressão regular contém tanto subexpressões vorazes quanto não vorazes, o comprimento total da correspondência é o mais longo possível ou o mais curto possível, de acordo com o atributo atribuído a expressão regular como um todo. Os atributos atribuídos às subexpressões afetam somente quanto desta correspondência estas podem “comer” com relação a cada uma outra.

Os quantificadores  $\{1, 1\}$  e  $\{1, 1\}?$  podem ser utilizados para obrigar voracidade e não-voracidade, respectivamente, nas subexpressões ou na expressão regular como um todo.

Os comprimentos de correspondência são medidos em caracteres, e não em elementos de classificação. Uma cadeia de caracteres vazia é considerada mais longa do que nenhuma correspondência. Por exemplo:  $bb^*$  corresponde aos três caracteres do meio de  $abbbc$ ;  $(week|wee)(night|knights)$  corresponde a todos os dez caracteres de  $weeknights$ ; quando é feita a correspondência entre  $(.)*.$  e  $abc$ , a subexpressão entre parênteses corresponde a todos os três caracteres; e quando é feita a correspondência entre  $(a^*)^*$  e  $bc$ , tanto toda a expressão regular quanto a subexpressão entre parênteses correspondem a uma cadeia de caracteres vazia.

Se for especificada uma correspondência não diferenciando maiúsculas de minúsculas, o efeito é como se toda a distinção entre letras maiúsculas e minúsculas tivesse desaparecido do alfabeto. Quando um caractere alfabético que existe em minúscula e maiúscula aparece como um caractere comum fora da expressão de colchetes, este é efetivamente transformado em uma expressão de colchetes contendo as duas representações: por exemplo,  $x$  se torna  $[xX]$ . Quando aparece dentro de uma expressão de colchetes, todas as outras representações são adicionadas à expressão de colchetes: por exemplo,  $[x]$  se torna  $[xX]$  e  $[\^x]$  se torna  $[\^xX]$ .

Se for especificada uma correspondência sensível a nova-linha, o `.` e as expressões de colchetes utilizando `^` nunca vão corresponder ao caractere de nova-linha (portanto a correspondência nunca atravessa as novas-linhas, a menos que a expressão regular determine explicitamente que isto seja feito), e `^` e `$` vão corresponder à cadeia de caracteres vazia após e antes da nova-linha, respectivamente, além de corresponderem ao início e fim da cadeia de caracteres, respectivamente. Mas os escapes `\A` e `\Z` das ERAs continuam a corresponder *apenas* ao início e fim da cadeia de caracteres.

Se for especificada a correspondência sensível a nova-linha parcial, isto afeta o `.` e as expressões de colchetes como na correspondência sensível a nova-linha, mas não o `^` e o `$`.

Se for especificada a correspondência sensível a nova-linha parcial inversa, isto afeta o `^` e o `$` como na correspondência sensível a nova-linha, mas não o `.` e as expressões de colchetes. Não é muito útil mas é fornecido para simetria.

### 9.7.3.6. Limites e compatibilidade

Nenhum limite específico é imposto para o comprimento das expressões regulares nesta implementação. Entretanto, os programas que pretendem ser altamente portáteis não devem utilizar expressões regulares mais longas do que 256 bytes, porque uma implementação em conformidade com o POSIX pode não aceitar estas expressões regulares.

A única funcionalidade das ERAs realmente incompatível com as EREs do POSIX, é que a `\` não perde seu significado especial dentro das expressões de colchetes. Todas as outras funcionalidades das ERAs utilizam uma sintaxe que é ilegal ou possuem efeitos não definidos ou não especificados nas EREs POSIX; a sintaxe de diretrizes na forma `***` está fora da sintaxe POSIX tanto para as ERBs quanto para as EREs.

Muitas extensões das ERAs foram pegadas emprestadas do Perl, mas algumas foram modificadas para que ficassem limpas, e umas poucas extensões do Perl não estão presentes. Entre as incompatibilidades a serem notadas estão `\b`, `\B`, a falta de um tratamento especial para a nova-linha final, a adição de expressões de colchetes complementadas para as coisas afetadas pela correspondência sensível à nova-linha, as restrições nos parênteses e referências para trás nas restrições de procura adiante, e a semântica de correspondência contendo correspondência mais longa/mais curta (em vez de primeira correspondência).

Existem duas incompatibilidades significativas entre as sintaxes das ERAs e das ERE reconhecida pelas versões do PostgreSQL anteriores a 7.4:

- Nas ERAs, a `\` seguida por um caractere alfanumérico é um escape ou um erro, enquanto nas versões anteriores, era apenas uma outra forma de escrever o caractere alfanumérico. Isto não deve causar problemas, porque não havia razão para escrever esta sequência nas versões anteriores.
- Nas ERAs, a `\` permanece sendo um caractere especial dentro de `[]`, portanto o literal `\` dentro da expressão de colchetes deve ser escrito como `\\`.

Embora estas diferenças provavelmente não devam criar problemas para a maioria dos aplicativos, podem ser evitadas se for necessário definindo `regex_flavor` como `extended`.

### 9.7.3.7. Expressões regulares básicas

As ERBs diferem das EREs sob vários aspectos. `|`, `+` e `?` são caracteres comuns e não existe nada equivalente para suas funcionalidades. Os delimitadores para limites são `\{` e `\}`, com `{` e `}` por si só sendo caracteres comuns. Os parênteses para as subexpressões aninhadas são `\(` e `\)`, com `(` e `)` por si só sendo caracteres comuns. `^` é um caractere comum exceto no início da expressão regular ou no início de uma subexpressão entre parênteses, `$` é um caractere comum exceto no final da expressão regular ou no final de uma subexpressão entre parênteses, e `*` é um caractere comum se aparecer no início da expressão regular ou no início de uma subexpressão entre parênteses, (após um possível `^` de abertura). Por fim, referências para trás de um único dígito estão disponíveis, e `\<` e `\>` são sinônimos de `[[ :< : ]]` e `[[ :> : ]]` respectivamente; nenhum outro escape está disponível.

## 9.7.4. Exemplos

**Nota:** Seção escrita pelo tradutor, não fazendo parte do manual original.

### Exemplo 9-4. Utilização de expressão regular em consulta

Neste exemplo são utilizadas expressões regulares em consultas a uma tabela. São mostradas, também, consultas semelhantes utilizando a função `REGEXP_LIKE` do Oracle 10g, para efeitos de comparação.<sup>5</sup>

Deve ser observado que no PostgreSQL quando se usa `SIMILAR TO` deve haver caractere de escape antes do sublinhado que não é metacaractere (“\\_”), mesmo entre colchetes, enquanto quando se usa o operador “~” deve haver caractere de escape antes do ponto que não é metacaractere (“\.”), ou ser colocado entre colchetes (“[.]”) para que este perca o seu significado especial. O PostgreSQL utiliza a função `similar_escape()` para converter as expressões regulares no estilo SQL:1999 para o estilo POSIX, para poder usá-las em seu processador de expressões regulares.

A função `REGEXP_LIKE` do Oracle 10g é semelhante ao operador “~” do PostgreSQL, mas enquanto na função `REGEXP_LIKE` é necessária apenas uma contrabarra para o escape de metacaracteres, no operador “~” são necessárias duas contrabarras, porque a primeira é consumida pelo interpretador de literais cadeia de caracteres. Nos dois casos, como é habitual nas expressões regulares, a correspondência se dá em qualquer ponto, a não ser que sejam utilizados os metacaracteres `^` e `$` para ancorar, explicitamente, a correspondência no início ou no fim da cadeia de caracteres, respectivamente.

Abaixo está mostrado o script usado para criar e carregar a tabela:

```
\!chcp 1252
CREATE TABLE textos(texto VARCHAR(40));
INSERT INTO textos VALUES ('www.apache.org');
INSERT INTO textos VALUES ('pgdocptbr.sourceforge.net');
INSERT INTO textos VALUES ('WWW.PHP.NET');
INSERT INTO textos VALUES ('www-130.ibm.com');
INSERT INTO textos VALUES ('Julia Margaret Cameron');
INSERT INTO textos VALUES ('Sor Juana Inés de la Cruz');
INSERT INTO textos VALUES ('Inês Pedrosa');
INSERT INTO textos VALUES ('Amy Semple McPherson');
INSERT INTO textos VALUES ('Mary McCarthy');
INSERT INTO textos VALUES ('Isabella Andreine');
INSERT INTO textos VALUES ('Jeanne Marie Bouvier de la Motte Guyon');
INSERT INTO textos VALUES ('Maria Tinteretto');
INSERT INTO textos VALUES ('');
INSERT INTO textos VALUES (' ' || chr(9) || chr(10) || chr(11) || chr(12) || chr(13));
INSERT INTO textos VALUES ('192.168.0.15');
INSERT INTO textos VALUES ('pgsql-bugs-owner@postgresql.org');
INSERT INTO textos VALUES('00:08:54:15:E5:FB');
```

A seguir estão mostradas as consultas efetuadas juntamente com seus resultados:

- I. Selecionar textos contendo um ou mais caracteres de “a” até “z”, seguidos por um ponto, seguido por um ou mais caracteres de “a” até “z”, seguidos por um ponto, seguido por um ou mais caracteres de “a” até “z”.

- PostgreSQL 8.0.0

```
=> SELECT texto FROM textos WHERE texto SIMILAR TO '([a-z]+).([a-z]+).([a-z]+)';
-- ou
=> SELECT texto FROM textos WHERE texto ~ '^([a-z]+)\.([a-z]+)\.([a-z]+)$';
-- ou
=> SELECT texto FROM textos WHERE texto ~ '^([a-z]+)[.](a-z+)[.](a-z+)$';
```

```
      texto
-----
www.apache.org
pgdocptbr.sourceforge.net
(2 linhas)
```

- Oracle 10g

```
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto, '^([a-z]+)\.([a-z]+)\.([a-z]+)$');
-- ou
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto, '^([a-z]+)[.](a-z+)[.](a-z+)$');
```

```
TEXTTO
-----
www.apache.org
pgdocptbr.sourceforge.net
```

II. Selecionar textos contendo um ou mais caracteres alfanuméricos, sublinhado ou hífen, seguidos por um ponto, seguido por um ou mais caracteres alfanuméricos, sublinhado ou hífen, seguidos por um ponto, seguido por um ou mais caracteres alfanuméricos ou sublinhado (\w corresponde a [a-zA-Z0-9\_], ou seja, alfanuméricos e sublinhado).

- PostgreSQL 8.0.0

```
=> SELECT texto
-> FROM textos
-> WHERE texto SIMILAR TO '([[:alnum:]]\[_-]+\).([[:alnum:]]\[_-]+\).([[:alnum:]]\[_-]+)';
-- ou
-> WHERE texto SIMILAR TO '([\w-]+\).([\w-]+\).([\w-]+)';
-- ou
-> WHERE texto ~ '^([[:alnum:]]\[_-]+\)\.([[:alnum:]]\[_-]+\)\.([[:alnum:]]\[_-]+)$';
-- ou
-> WHERE texto ~ '^(([[:alnum:]]\[_-]+\)\.){2}([[:alnum:]]\[_-]+)$';
-- ou
-> WHERE texto ~ '^([\w-]+\)\.([\w-]+\)\.([\w-]+)$';
-- ou
-> WHERE texto ~ '^(([\\w-]+\)\.){2}([\\w-]+)$';
```

```

      texto
-----
www.apache.org
pgdocptbr.sourceforge.net
WWW.PHP.NET
www-130.ibm.com
(4 linhas)
```

- Oracle 10g

```
SQL> SELECT texto
      2 FROM textos
      3 WHERE REGEXP_LIKE(texto, '^([[:alnum:]]\[_-]+\).([[:alnum:]]\[_-]+\).([[:alnum:]]\[_-]+)$');
```

```

TEXT0
-----
www.apache.org
pgdocptbr.sourceforge.net
WWW.PHP.NET
www-130.ibm.com
```

III. Selecionar textos começando pela letra “S” maiúscula (\_\* equivale a %).

- PostgreSQL 8.0.0

```
=> SELECT texto FROM textos WHERE texto SIMILAR TO 'S%';
-- ou
=> SELECT texto FROM textos WHERE texto SIMILAR TO 'S_*';
-- ou
=> SELECT texto FROM textos WHERE texto ~ '^S';
```

```

      texto
-----
Sor Juana Inés de la Cruz
(1 linha)
```

- Oracle 10g

```
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto, '^S');
```

```

TEXT0
-----
Sor Juana Inés de la Cruz
```

IV. Selecionar textos contendo “Mc” em qualquer posição.

- PostgreSQL 8.0.0

```
=> SELECT texto FROM textos WHERE texto SIMILAR TO '%Mc%';
-- ou
```



```
=> SELECT texto FROM textos WHERE texto ~ 'Mc';
```

```

      texto
-----
Amy Semple McPherson
Mary McCarthy
(2 linhas)

```

- Oracle 10g

```
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto, 'Mc');
```

```

TEXT0
-----
Amy Semple McPherson
Mary McCarthy

```

## V. Selecionar textos terminados por “on”.

- PostgreSQL 8.0.0

```
=> SELECT texto FROM textos WHERE texto SIMILAR TO '%on';
-- ou
=> SELECT texto FROM textos WHERE texto ~ 'on$';
```

```

      texto
-----
Julia Margaret Cameron
Amy Semple McPherson
Jeanne Marie Bouvier de la Motte Guyon
(3 linhas)

```

- Oracle 10g

```
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto, 'on$');
```

```

TEXT0
-----
Julia Margaret Cameron
Amy Semple McPherson
Jeanne Marie Bouvier de la Motte Guyon

```

## VI. Selecionar textos começando por “J” e terminando por “n”.

- PostgreSQL 8.0.0

```
=> SELECT texto FROM textos WHERE texto SIMILAR TO 'J%n';
-- ou
=> SELECT texto FROM textos WHERE texto ~ '^J.*n$';
```

```

      texto
-----
Julia Margaret Cameron
Jeanne Marie Bouvier de la Motte Guyon
(2 linhas)

```

- Oracle 10g

```
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto, '^J.*n$');
```

```

TEXT0
-----
Julia Margaret Cameron
Jeanne Marie Bouvier de la Motte Guyon

```

## VII. Selecionar textos contendo “Cameron” ou “Marie” em qualquer posição.

- PostgreSQL 8.0.0

```
=> SELECT texto FROM textos WHERE texto SIMILAR TO '%(Cameron|Marie)%';
-- ou
=> SELECT texto FROM textos WHERE texto ~ '(Cameron|Marie)';
```

```

      texto
-----
Julia Margaret Cameron
Jeanne Marie Bouvier de la Motte Guyon
(2 linhas)

```

- Oracle 10g

```
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto, '(Cameron|Marie)');
```

```

TEXT0
-----
Julia Margaret Cameron
Jeanne Marie Bouvier de la Motte Guyon

```

### VIII. Selecionar textos contendo “Maria”, “Marie” ou “Mary” (não seleciona “Margaret”).

- PostgreSQL 8.0.0

```
=> SELECT texto FROM textos WHERE texto SIMILAR TO '%Mar(ia|ie|y)%';
```

```
-- ou
```

```
=> SELECT texto FROM textos WHERE texto ~ 'Mar(ia|ie|y)';
```

```

      texto
-----
Mary McCarthy
Jeanne Marie Bouvier de la Motte Guyon
Maria Tinteretto
(3 linhas)

```

- Oracle 10g

```
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto, 'Mar(ia|ie|y)');
```

```

TEXT0
-----
Mary McCarthy
Jeanne Marie Bouvier de la Motte Guyon
Maria Tinteretto

```

### IX. Selecionar textos sem nenhum caractere, ou com somente espaços em branco (no Oracle texto sem nenhum caractere é igual a nulo).

- PostgreSQL 8.0.0

```
=> SELECT texto, length(texto) FROM textos WHERE texto SIMILAR TO '[:space:]*';
```

```
-- ou
```

```
=> SELECT texto, length(texto) FROM textos WHERE texto SIMILAR TO '\\s*';
```

```
-- ou
```

```
=> SELECT texto, length(texto) FROM textos WHERE texto ~ '^[:space:]*$';
```

```
-- ou
```

```
=> SELECT texto, length(texto) FROM textos WHERE texto ~ '^\\s*$';
```

```

texto | length
-----+-----
      |      0
      |      7
(2 linhas)

```

- Oracle 10g

```
SQL> SELECT texto, length(texto) FROM textos WHERE REGEXP_LIKE(texto, '^[:space:]*$');
```

```

TEXT0                                LENGTH(TEXT0)
-----

```

X. Selecionar textos contendo a letra “W” minúscula ou maiúscula, seguida por uma ou mais letras “W” minúsculas ou maiúsculas.

- PostgreSQL 8.0.0

```
=> SELECT texto FROM textos WHERE texto SIMILAR TO '%[Ww]([Ww]+)%';
-- ou
=> SELECT texto FROM textos WHERE texto ~ '[Ww]([Ww]+)';
-- ou
=> SELECT texto FROM textos WHERE texto ~ '[Ww]{2,}';
```

```
      texto
-----
www.apache.org
WWW.PHP.NET
www-130.ibm.com
(3 linhas)
```

- Oracle 10g

```
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto, '[Ww]([Ww]+)');
-- ou
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto, '[Ww]{2,}');
```

```
      TEXTO
-----
www.apache.org
WWW.PHP.NET
www-130.ibm.com
```

XI. Selecionar textos contendo “Inês”, com letras maiúsculas ou minúsculas, com ou sem acento, em qualquer posição.

- PostgreSQL 8.0.0

```
=> SELECT texto FROM textos WHERE texto SIMILAR TO '%[Ii][nN][eéêÊÊÊ][Ss]%';
-- ou
=> SELECT texto FROM textos WHERE lower(to_ascii(texto)) SIMILAR TO '%ines%';
-- ou
=> SELECT texto FROM textos WHERE texto ~ '[Ii][nN][eéêÊÊÊ][Ss]';
-- ou
=> SELECT texto FROM textos WHERE lower(to_ascii(texto)) ~ 'ines';
```

```
      texto
-----
Sor Juana Inés de la Cruz
Inês Pedrosa
(2 linhas)
```

- Oracle 10g

```
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto, '[Ii][nN][eéêÊÊÊ][Ss]');
```

```
      TEXTO
-----
Sor Juana Inés de la Cruz
Inês Pedrosa
```

XII. Selecionar textos contendo dígitos.

- PostgreSQL 8.0.0

```
=> SELECT texto FROM textos WHERE texto SIMILAR TO '%[0-9]+%';
-- ou
=> SELECT texto FROM textos WHERE texto SIMILAR TO '%[:digit:]+%';
-- ou
=> SELECT texto FROM textos WHERE texto SIMILAR TO '%\\d+%';
-- ou
=> SELECT texto FROM textos WHERE texto ~ '[0-9]+';
-- ou
=> SELECT texto FROM textos WHERE texto ~ '[:digit:]+';
```

```
-- ou
=> SELECT texto FROM textos WHERE texto ~ '\\d+';
```

```

      texto
-----
www-130.ibm.com
192.168.0.15
00:08:54:15:E5:FB
(3 linhas)
```

- Oracle 10g

```
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto, '[0-9]+');
-- ou
SQL> SELECT texto FROM textos WHERE REGEXP_LIKE(texto, '[:digit:]+');
```

```

TEXT0
-----
www-130.ibm.com
192.168.0.15
00:08:54:15:E5:FB
```

XIII. Selecionar textos não contendo espaços em branco (texto sem nenhum caractere não possui espaço em branco, mas no Oracle texto sem nenhum caractere é igual a nulo).

- PostgreSQL 8.0.0

```
=> SELECT texto FROM textos WHERE texto NOT SIMILAR TO '%\\s%';
-- ou
=> SELECT texto FROM textos WHERE texto NOT SIMILAR TO '%[:space:]]%';
-- ou
=> SELECT texto FROM textos WHERE texto !~ '\\s';
-- ou
=> SELECT texto FROM textos WHERE texto !~ '[:space:]]';
```

```

      texto
-----
www.apache.org
pgdocptbr.sourceforge.net
WWW.PHP.NET
www-130.ibm.com

192.168.0.15
pgsql-bugs-owner@postgresql.org
00:08:54:15:E5:FB
(8 linhas)
```

- Oracle 10g

```
SQL> SELECT texto FROM textos WHERE NOT REGEXP_LIKE(texto, '[:space:]]');
```

```

TEXT0
-----
www.apache.org
pgdocptbr.sourceforge.net
WWW.PHP.NET
www-130.ibm.com
192.168.0.15
pgsql-bugs-owner@postgresql.org
00:08:54:15:E5:FB
```

XIV. Selecionar textos que podem ser endereços de IPv4.

- PostgreSQL 8.0.0

```
=> SELECT texto
-> FROM textos
```

```

->                                WHERE                                texto                                ~
'^[[:digit:]]{1,3}[.][[:digit:]]{1,3}[.][[:digit:]]{1,3}[.][[:digit:]]{1,3}$';
-- ou
-> WHERE texto ~ '^([[:digit:]]{1,3}[.]){3}[[:digit:]]{1,3}$';

      texto
-----
192.168.0.15
(1 linha)

```

- Oracle 10g

```

SQL> SELECT texto
      2 FROM textos
      3                                WHERE                                REGEXP_LIKE(texto,
'^[[:digit:]]{1,3}[.][[:digit:]]{1,3}[.][[:digit:]]{1,3}[.][[:digit:]]{1,3}$');
-- ou
      3 WHERE REGEXP_LIKE(texto, '^([[:digit:]]{1,3}[.]){3}[[:digit:]]{1,3}$');

TEXTO
-----
192.168.0.15

```

#### XV. Selecionar textos que podem ser endereços de correio eletrônico.

- PostgreSQL 8.0.0

```

=> SELECT texto
-> FROM textos
-> WHERE texto SIMILAR TO '[a-zA-Z][\w.-]*@[a-zA-Z][\w.-]*.[a-zA-Z]+';
-- ou
-> WHERE texto SIMILAR TO '[a-zA-Z][[:alnum:]\_.-]*@[a-zA-Z][[:alnum:]\_.-]*.[a-zA-Z]+';
-- ou
-> WHERE texto ~ '^[a-zA-Z][\w.-]*@[a-zA-Z][\w.-]*[.][a-zA-Z]+';
-- ou
-> WHERE texto ~ '^[a-zA-Z][[:alnum:]\_.-]*@[a-zA-Z][[:alnum:]\_.-]*[.][a-zA-Z]+';

      texto
-----
pgsql-bugs-owner@postgresql.org
(1 linha)

```

- Oracle 10g

```

SQL> SELECT texto
      2 FROM textos
      3 WHERE REGEXP_LIKE(texto, '^([[:alnum:]\_.-]+@[[:alnum:]\_.-]+\.[a-zA-Z]+)$');

TEXTO
-----
pgsql-bugs-owner@postgresql.org

```

#### XVI. Selecionar textos que podem ser endereços de hardware de placa Ethernet, formados por 48 bits, expressos como 12 dígitos hexadecimais (0-9 e A-F maiúsculas), no formato 12:34:56:78:9A:BC.

- PostgreSQL 8.0.0

```

=> SELECT texto
-> FROM textos
-> WHERE texto ~ '^([0-9A-F]{2}:){5}[0-9A-F]{2}$';
-- ou
-> WHERE texto ~ '^([0-9A-F]{2}:){5}[0-9A-F]{2}$';

      texto
-----
00:08:54:15:E5:FB

```

```
(1 linha)

• Oracle 10g
SQL> SELECT texto
      2 FROM textos
      3 WHERE REGEXP_LIKE(texto, '^([0-9A-F]{2}:){5}[0-9A-F]{2}$');
-- ou
      3 WHERE REGEXP_LIKE(texto, '^([0-9A-F]{2}:){5}[0-9A-F]{2}$');

TEXT0
-----
00:08:54:15:E5:FB
```

### Exemplo 9-5. Utilização de expressão regular em restrição de verificação

Neste exemplo são utilizadas expressões regulares em restrição de verificação na criação de uma tabela. São utilizadas expressões regulares para verificar se os dados fornecidos para os campos de endereço de correio eletrônico, CEP de 9 dígitos com hífen, CEP de 8 dígitos sem hífen e unidade da federação são válidos. É mostrada, também, a criação de uma tabela semelhantes utilizando a função `REGEXP_LIKE` do Oracle 10g na restrição de verificação, para efeitos de comparação.<sup>6</sup>

Deve ser observado que no Oracle 10g só é necessário utilizar uma contrabarra antes do ponto, e não duas como no PostgreSQL 8.0.0.

Abaixo está mostrada a execução do exemplo no PostgreSQL 8.0.0:

```
=> CREATE TABLE contactos(
(>   email VARCHAR(40)    -- endereço de correio eletrônico
(>       CONSTRAINT chk_email
(>       CHECK (email ~ '^([a-zA-Z][[:alnum:]]_.-)*@[a-zA-Z][[:alnum:]]_.-*.[a-zA-Z]+$',
(>   cep8 CHAR(8)          -- CEP com oito dígitos sem hífen
(>       CONSTRAINT chk_cep8
(>       CHECK (cep8 ~ '^([[:digit:]]{8})$',
(>   cep9 CHAR(9)          -- CEP com nove dígitos com hífen
(>       CONSTRAINT chk_cep9
(>       CHECK (cep9 ~ '^([[:digit:]]{5}-[[:digit:]]{3})$',
(>   uf CHAR(2)            -- unidade da federação
(>       CONSTRAINT chk_uf
(>       CHECK (uf ~
'^A(C|L|M|P)|B(A|C|E|D|F|E|S|G|O|M(A|G|S|T)|P(A|B|E|I|R)|R(J|N|O|R|S)|S(C|E|P)|TO$')
(> );

=> INSERT INTO contactos VALUES('clp.decom@camara.gov.br','70160900','70160-900','DF');
=> INSERT INTO contactos VALUES('clp.decomcamara.gov.br','70160900','70160-900','DF');
ERRO: a nova linha para a relação "contactos" viola a restrição de verificação "chk_email"
=> INSERT INTO contactos VALUES('clp.decom@camara.gov.br','7016090','70160-900','DF');
ERRO: a nova linha para a relação "contactos" viola a restrição de verificação "chk_cep8"
=> INSERT INTO contactos VALUES('clp.decom@camara.gov.br','70160900','70160900','DF');
ERRO: a nova linha para a relação "contactos" viola a restrição de verificação "chk_cep9"
=> INSERT INTO contactos VALUES('clp.decom@camara.gov.br','70160900','70160-900','DG');
ERRO: a nova linha para a relação "contactos" viola a restrição de verificação "chk_uf"

=> SELECT * FROM contactos;

      email      | cep8 | cep9 | uf
-----+-----+-----+---
 clp.decom@camara.gov.br | 70160900 | 70160-900 | DF
(1 linha)
```

Abaixo está mostrada a execução de exemplo semelhante no Oracle 10g:

```
SQL> CREATE TABLE contactos(
      2   email VARCHAR(40)    -- endereço de correio eletrônico
```

```

3      CONSTRAINT chk_email
4      CHECK (REGEXP_LIKE(email, '^([a-zA-Z][[:alnum:]]_.-)*@[a-zA-Z][[:alnum:]]_.-
]*[.][a-zA-Z]+$')),
5      cep8 CHAR(8)          -- CEP com oito dígitos sem hífen
6      CONSTRAINT chk_cep8
7      CHECK (REGEXP_LIKE(cep8, '^([[:digit:]]{8}$')),
8      cep9 CHAR(9)          -- CEP com nove dígitos com hífen
9      CONSTRAINT chk_cep9
10     CHECK (REGEXP_LIKE(cep9, '^([[:digit:]]{5}-[[:digit:]]{3}$')),
11     uf CHAR(2)             -- unidade da federação
12     CONSTRAINT chk_uf
13     CHECK (REGEXP_LIKE(uf,
14     '^A(C|L|M|P)|BA|CE|DF|ES|GO|M(A|G|S|T)|P(A|B|E|I|R)|R(J|N|O|R|S)|S(C|E|P)|TO$'));

```

```
SQL> INSERT INTO contactos VALUES('clp.decom@camara.gov.br','70160900','70160-900','DF');
```

```
SQL> INSERT INTO contactos VALUES('clp.decomcamara.gov.br','70160900','70160-900','DF');
```

```
INSERT INTO contactos VALUES('clp.decomcamara.gov.br','70160900','70160-900','DF')
```

```
*
```

```
ERRO na linha 1:
```

```
ORA-02290: restrição de verificação (SCOTT.CHK_EMAIL) violada
```

```
SQL> INSERT INTO contactos VALUES('clp.decom@camara.gov.br','7016090','70160-900','DF');
```

```
INSERT INTO contactos VALUES('clp.decom@camara.gov.br','7016090','70160-900','DF')
```

```
*
```

```
ERRO na linha 1:
```

```
ORA-02290: restrição de verificação (SCOTT.CHK_CEP8) violada
```

```
SQL> INSERT INTO contactos VALUES('clp.decom@camara.gov.br','70160900','70160900','DF');
```

```
INSERT INTO contactos VALUES('clp.decom@camara.gov.br','70160900','70160900','DF')
```

```
*
```

```
ERRO na linha 1:
```

```
ORA-02290: restrição de verificação (SCOTT.CHK_CEP9) violada
```

```
SQL> INSERT INTO contactos VALUES('clp.decom@camara.gov.br','70160900','70160-900','DG');
```

```
INSERT INTO contactos VALUES('clp.decom@camara.gov.br','70160900','70160-900','DG')
```

```
*
```

```
ERRO na linha 1:
```

```
ORA-02290: restrição de verificação (SCOTT.CHK_UF) violada
```

```
SQL> SELECT * FROM contactos;
```

EMAIL	CEP8	CEP9	UF
clp.decom@camara.gov.br	70160900	70160-900	DF

### Exemplo 9-6. Utilização de expressão regular em colunas concatenadas

Neste exemplo é utilizada uma expressão regular na consulta a uma tabela, onde o primeiro nome e o sobrenome estão armazenados em colunas diferentes, para localizar o nome de uma pessoa que pode estar armazenado de várias formas diferentes. Abaixo está mostrado o script usado para criar e carregar a tabela:

```

CREATE TEMPORARY TABLE nomes (nome TEXT, sobrenome TEXT);
INSERT INTO nomes VALUES ('Manuel','da Paróquia');
INSERT INTO nomes VALUES ('SEU MANUEL','DA PAROQUIA');
INSERT INTO nomes VALUES ('Manuel','da Paroquia dos Anjos');
INSERT INTO nomes VALUES ('Manuel da Paróquia',NULL);
INSERT INTO nomes VALUES ('Seu','MANUEL DA PARÓQUIA');
INSERT INTO nomes VALUES ('Dona Maria','da Capela');

```

Abaixo está mostrada a consulta realizada sobre os campos nome e sobrenome concatenados, para localizar uma pessoa com “Manuel” no nome e “Paróquia” no sobrenome, juntamente com os resultados obtidos.

```
=> \pset null '(nulo)'
A visualização de nulos é "(nulo)".
=> SELECT *
-> FROM nomes
-> WHERE (COALESCE(nome, '') || COALESCE(sobrenome, '')) ~* 'manuel.*par[oó]quia';
```

nome	sobrenome
Manuel	da Paróquia
SEU MANUEL	DA PAROQUIA
Manuel	da Paroquia dos Anjos
Manuel da Paróquia	(nulo)
Seu	MANUEL DA PARÓQUIA

(5 linhas)

### Exemplo 9-7. Comparação entre o PostgreSQL e o PHP

Neste exemplo é feita a comparação entre a função `substring` do PostgreSQL e a função `preg_match` do PHP, utilizando uma expressão regular voraz e outra não voraz, para capturar o texto entre as marcas `<a...>` e `</a>` na cadeia de caracteres:

```
O <a href="http://php.net/">PHP</a> possui um excelente <a
href="http://php.net/manual">manual</a>.
```

Os resultados são idênticos, mas a função `preg_match` retorna uma matriz onde o elemento `[0]` contém o texto que corresponde a todo o padrão, o elemento `[1]` contém o texto que corresponde à primeira subexpressão entre parênteses capturada, e assim por diante, enquanto a função `substring` retorna o texto que corresponde à primeira subexpressão entre parênteses apenas.<sup>7</sup>

- PostgreSQL 8.0.0

```
=> \set texto '\O <a href="http://php.net/">PHP</a> possui um excelente '
=> \set texto :texto '<a href="http://php.net/manual">manual</a>.\''
```

```
=> SELECT substring(:texto, '<a.*>(.)</a>');
```

```
substring
-----
manual
(1 linha)
```

```
=> SELECT substring(:texto, '<a.*?>(.*?)</a>');
```

```
substring
-----
PHP
(1 linha)
```

- PHP

```
<?php
    $texto = 'O <a href="http://php.net/">PHP</a> possui um excelente '
        . '<a href="http://php.net/manual">manual</a>.';
    preg_match('<a.*>(.)</a>|', $texto, $corresp);
    print_r($corresp);
    preg_match('<a.*?>(.*?)</a>|', $texto, $corresp);
    print_r($corresp);
?>
```

```
Array
(
    [0] => <a href="http://php.net/">PHP</a> possui um excelente <a
href="http://php.net/manual">manual</a>
    [1] => manual
)
```

```
Array
(
    [0] => <a href="http://php.net/">PHP</a>
    [1] => PHP
)
```



)

### Exemplo 9-8. Comparação entre o PostgreSQL e o PCRE

Neste exemplo é feita a comparação entre a função `substring` do PostgreSQL e o programa `pcredemo` do PCRE - Perl Compatible Regular Expressions (<http://www.pcre.org/>), utilizando uma expressão regular voraz e outra não voraz, para capturar o texto entre as marcas `<a...>` e `</a>` na cadeia de caracteres:

```
O      <a      href="http://php.net/">PHP</a>      possui      um      excelente      <a
href="http://php.net/manual">manual</a>.
```

Os resultados são idênticos, mas o programa `pcredemo` com a opção `-g` retorna todas as correspondências encontradas, enquanto a função `substring` retorna o texto que corresponde à primeira subexpressão entre parênteses apenas. Sem a opção `-g`, o resultado do `pcredemo` é idêntico ao do PHP ([http://br.php.net/manual/pt\\_BR/ref.pcre.php](http://br.php.net/manual/pt_BR/ref.pcre.php)), que utiliza a biblioteca PCRE como suporte às expressões regulares.<sup>8</sup>

- PostgreSQL 8.0.0

```
=> \set texto '\O <a href="http://php.net/">PHP</a> possui um excelente '
```

```
=> \set texto :texto '<a href="http://php.net/manual">manual</a>.\''
```

```
=> SELECT substring(:texto, '<a.*>(.*?)</a>');
```

```
substring
-----
manual
(1 linha)
```

```
=> SELECT substring(:texto, '<a.*?>(.*?)</a>');
```

```
substring
-----
PHP
(1 linha)
```

- PCRE

```
$ ./pcredemo -g \
```

```
> '<a.*>(.*?)</a>' \
```

```
> 'O <a href="http://php.net/">PHP</a> possui um excelente <a
href="http://php.net/manual">manual</a>.'
```

Correspondência bem sucedida no deslocamento 2

```
0: <a href="http://php.net/">PHP</a> possui um excelente <a href="http://php.net/manual">manual</a>
1: manual
```

```
$ ./pcredemo -g \
```

```
> '<a.*?>(.*?)</a>' \
```

```
> 'O <a href="http://php.net/">PHP</a> possui um excelente <a
href="http://php.net/manual">manual</a>.'
```

Correspondência bem sucedida no deslocamento 2

```
0: <a href="http://php.net/">PHP</a>
1: PHP
```

Correspondência bem sucedida novamente no deslocamento 56

```
0: <a href="http://php.net/manual">manual</a>
1: manual
```

## 9.8. Funções para formatar tipo de dado

As funções de formatação do PostgreSQL fornecem um poderoso conjunto de ferramentas para converter vários tipos de dado (`date/time`, `integer`, `floating point`, `numeric`) em cadeias de caracteres formatadas, e para converter cadeias de caracteres formatadas em tipos de dado específicos. A Tabela 9-22 mostra estas funções, que seguem uma convenção de chamada comum: o primeiro argumento é o valor a ser formatado, e o segundo argumento é o modelo que define o formato da entrada ou da saída.

**Tabela 9-22. Funções de formatação**

Função	Tipo retornado	Descrição	Exemplo
<code>to_char(timestamp, text)</code>	text	converte carimbo do tempo (time stamp) em cadeia de caracteres	<code>to_char(current_timestamp, 'HH12:MI:SS')</code>
<code>to_char(interval, text)</code>	text	converte intervalo em cadeia de caracteres	<code>to_char(interval '15h 2m 12s', 'HH24:MI:SS')</code>
<code>to_char(int, text)</code>	text	converte inteiro em cadeia de caracteres	<code>to_char(125, '999')</code>
<code>to_char(double precision, text)</code>	text	converte real e precisão dupla em cadeia de caracteres	<code>to_char(125.8::real, '999D9')</code>
<code>to_char(numeric, text)</code>	text	converte numérico em cadeia de caracteres	<code>to_char(-125.8, '999D99S')</code>
<code>to_date(text, text)</code>	date	converte cadeia de caracteres em data	<code>to_date('05 Dec 2000', 'DD Mon YYYY')</code>
<code>to_timestamp(text, text)</code>	timestamp with time zone	converte cadeia de caracteres em carimbo do tempo	<code>to_timestamp('05 Dec 2000', 'DD Mon YYYY')</code>
<code>to_number(text, text)</code>	numeric	converte cadeia de caracteres em numérico	<code>to_number('12,454.8-', '99G999D9S')</code>

Advertência: `to_char(interval, text)` está obsoleta, não devendo ser utilizada nos novos aplicativos. Será removida na próxima versão.

Em uma cadeia de caracteres modelo de saída (para `to_char`), existem certos padrões que são reconhecidos e substituídos pelos dados devidamente formatados a partir do valor a ser formatado. Qualquer texto que não seja um modelo padrão é simplesmente copiado sem alteração. Da mesma forma, em uma cadeia de caracteres modelo de entrada (para qualquer coisa menos `to_char`), os modelos padrão identificam as partes da cadeia de caracteres da entrada de dados a serem procuradas, e os valores a serem encontrados nestas partes.

A Tabela 9-23 mostra os modelos padrão disponíveis para formatar valores de data e de hora.

**Tabela 9-23. Modelos padrão para formatação de data e hora**

Modelo	Descrição
HH	hora do dia (01-12)
HH12	hora do dia (01-12)
HH24	hora do dia (00-23)
MI	minuto (00-59)
SS	segundo (00-59)
MS	milissegundo (000-999)
US	microsegundo (000000-999999)
SSSS	segundos após a meia-noite (0-86399)
AM ou A.M. ou PM ou P.M.	indicador de meridiano (maiúsculas)

Modelo	Descrição
am ou a.m. ou pm ou p.m.	indicador de meridiano (minúsculas)
Y,YYY	ano (4 e mais dígitos) com vírgula
YYYY	ano (4 e mais dígitos)
YYY	últimos 3 dígitos do ano
YY	últimos 2 dígitos do ano
Y	último dígito do ano
IYYY	ano ISO (4 ou mais dígitos)
IYY	últimos 3 dígitos do ano ISO
IY	últimos 2 dígitos do ano ISO
I	último dígito do ano ISO
BC ou B.C. ou AD ou A.D.	indicador de era (maiúscula)
bc ou b.c. ou ad ou a.d.	indicador de era (minúscula)
MONTH	nome completo do mês em maiúsculas (9 caracteres completado com espaços)
Month	nome completo do mês em maiúsculas e minúsculas (9 caracteres completado com espaços)
month	nome completo do mês em minúsculas (9 caracteres completado com espaços)
MON	nome abreviado do mês em maiúsculas (3 caracteres)
Mon	nome abreviado do mês em maiúsculas e minúsculas (3 caracteres)
mon	nome abreviado do mês em minúsculas (3 caracteres)
MM	número do mês (01-12)
DAY	nome completo do dia em maiúsculas (9 caracteres completado com espaços)
Day	nome completo do dia em maiúsculas e minúsculas (9 caracteres completado com espaços)
day	nome completo do dia em minúsculas (9 caracteres completado com espaços)
DY	nome abreviado do dia em maiúsculas (3 caracteres)
Dy	nome abreviado do dia em maiúsculas e minúsculas (3 caracteres)
dy	nome abreviado do dia em minúsculas (3 caracteres)
DDD	dia do ano (001-366)
DD	dia do mês (01-31)
D	dia da semana (1-7; Domingo é 1)
W	semana do mês (1-5) onde a primeira semana começa no primeiro dia do mês
WW	número da semana do ano (1-53) onde a primeira semana começa no primeiro dia do ano
IW	número da semana do ano ISO (A primeira quinta-feira do novo ano está na semana 1)
CC	século (2 dígitos)

Modelo	Descrição
J	Dia Juliano (dias desde 1 de janeiro de 4712 AC)
Q	trimestre
RM	mês em algarismos romanos (I-XII; I=Janeiro) - maiúsculas
rm	mês em algarismos romanos (I-XII; I=Janeiro) - minúsculas
TZ	nome da zona horária - maiúsculas
tz	nome da zona horária - minúsculas

Certos modificadores podem ser aplicados aos modelos padrão para alterar seu comportamento. Por exemplo, `FMMonth` é o modelo “Month” com o modificador “FM”. A Tabela 9-24 mostra os modificadores de modelo para formatação de data e hora.

**Tabela 9-24. Modificadores de modelo padrão para formatação de data e hora**

Modificador	Descrição	Exemplo
prefixo FM	modo de preenchimento (suprime completar com brancos e zeros)	<code>FMMonth</code>
sufixo TH	sufixo de número ordinal maiúsculo	<code>DDTH</code>
sufixo th	sufixo de número ordinal minúsculo	<code>DDth</code>
prefixo FX	opção global de formato fixo (veja nota de utilização)	<code>FX Month DD Day</code>
sufixo SP	modo de falar ( <code>spell mode</code> ) (ainda não implementado)	<code>DDSP</code>

Notas sobre a utilização da formatação de data e hora:

- O FM suprime zeros à esquerda e espaços à direita, que de outra forma seriam adicionados para fazer a saída do modelo ter comprimento fixo.
- As funções `to_timestamp` e `to_date` saltam espaços em branco múltiplos na cadeia de caracteres de entrada quando a opção FX não é utilizada. O FX deve ser especificado como o primeiro item do modelo; por exemplo, `to_timestamp('2000 JUN', 'YYYY MON')` está correto, mas `to_timestamp('2000 JUN', 'FXYYYY MON')` retorna erro, porque `to_timestamp` espera um único espaço apenas.
- É permitida a presença de texto comum nos modelos para `to_char`, sendo mostrados literalmente na saída. Uma parte da cadeia de caracteres pode ser colocada entre aspas, para obrigar sua interpretação como um texto literal mesmo contendo palavras chave do modelo. Por exemplo, em `'"Hello Year "YYYY'`, o YYYY será substituído pelo ano do fornecido, mas o único Y em `Year` não será substituído.
- Se for desejada a presença de aspas na saída, as mesmas devem ser precedidas por contrabarra. Por exemplo `'\\"YYYY Month\\"'`. (Duas contrabarras são necessárias, porque a contrabarra possui significado especial em uma constante cadeia de caracteres).
- A conversão YYYY de cadeia de caracteres para `timestamp` ou para `date` tem restrição quando são utilizados anos com mais de 4 dígitos. Deve ser utilizado um modelo, ou algum caractere que não seja um dígito, após YYYY, senão o ano será sempre interpretado como tendo 4 dígitos. Por exemplo, (com o ano 20000): `to_date('200001121', 'YYYYMMDD')` é interpretado como um ano de 4 dígitos; em vez disso, deve ser utilizado um separador que não seja um dígito após o ano, como `to_date('20000-1121', 'YYYY-MMDD')` ou `to_date('20000Nov21', 'YYYYMonDD')`.
- Os valores de milissegundos MS e microssegundos US na conversão de uma cadeia de caracteres para um carimbo do tempo (`timestamp`), são interpretados como a sendo parte dos segundos após o ponto decimal. Por exemplo, `to_timestamp('12:3', 'SS:MS')` não são 3 milissegundos, mas 300, porque a conversão interpreta como sendo 12 + 0.3 segundos. Isto significa que, para o formato SS:MS, os valores de entrada 12:3, 12:30 e 12:300 especificam o mesmo número de milissegundos. Para especificar três milissegundos deve ser utilizado 12:003, que na conversão é interpretado como 12 + 0.003 = 12.003 segundos.

A seguir está mostrado um exemplo mais complexo:

`to_timestamp('15:12:02.020.001230', 'HH:MI:SS.MS.US')` é interpretado como 15 horas, 12 minutos e 2 segundos + 20 milissegundos + 1230 microssegundos = 2.021230 segundos.

- A numeração do dia da semana de `to_char` (veja o modelo padrão de formatação 'D') é diferente do dia da semana da função `extract`.

A Tabela 9-25 mostra os modelos padrão disponíveis para formatar valores numéricos.

**Tabela 9-25. Modelos padrão para formatação de números**

Modelo	Descrição
9	valor com o número especificado de dígitos
0	valor com zeros à esquerda
. (ponto)	ponto decimal
, (vírgula)	separador de grupo (milhares)
PR	valor negativo entre < e >
S	sinal preso ao número (utiliza o idioma)
L	símbolo da moeda (utiliza o idioma)
D	ponto decimal (utiliza o idioma)
G	separador de grupo (utiliza o idioma)
MI	sinal de menos na posição especificada (se número < 0)
PL	sinal de mais na posição especificada (se número > 0)
SG	sinal de mais/menos na posição especificada
RN <sup>a</sup>	algarismos romanos (entrada entre 1 e 3999)
TH ou th	sufixo de número ordinal
V	desloca o número especificado de dígitos (veja as notas sobre utilização)
EEEE	notação científica (ainda não implementada)
Notas:	
a. RN — roman numerals.	

Notas sobre a utilização da formatação numérica:

- O sinal formatado utilizando SG, PL ou MI não está ancorado ao número; por exemplo, `to_char(-12, 'S9999')` produz ' -12', mas `to_char(-12, 'MI9999')` produz '- 12'. A implementação do Oracle não permite utilizar o MI antes do 9, requerendo que o 9 preceda o MI.
- O 9 resulta em um valor com o mesmo número de dígitos que o número de 9s. Se não houver um dígito para colocar, é colocado espaço.
- O TH não converte valores menores que zero e não converte números fracionários.
- O PL, o SG e o TH são extensões do PostgreSQL.
- O V multiplica efetivamente os valores da entrada por  $10^n$ , onde  $n$  é o número de dígitos após o V. A função `to_char` não permite o uso de V junto com o ponto decimal (Por exemplo, `99.9V99` não é permitido).

A Tabela 9-26 mostra alguns exemplos de uso da função `to_char`.

Tabela 9-26. Exemplos de utilização da função to\_char

Expressão	PostgreSQL 8.0.0 <sup>a</sup>	Oracle 10g (N. do T.) <sup>b</sup>
to_char(current_timestamp, 'Day, DD HH12:MI:SS')	'Friday , 04 02:22:24'	'Friday , 04 02:22:24'
to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')	'Friday, 4 02:22:24'	'Friday, 04 02:22:24'
to_char(-0.1, '99.99')	' -.10'	' -.10'
to_char(-0.1, 'FM9.99')	'-.1'	'-.1'
to_char(0.1, '0.9')	' 0.1'	' 0.1'
to_char(12, '9990999.9')	' 0012.0'	' 0012.0'
to_char(12, 'FM9990999.9')	'0012.'	'0012.'
to_char(485, '999')	' 485'	' 485'
to_char(-485, '999')	'-485'	'-485'
to_char(485, '9 9 9')	' 4 8 5'	formato inválido
to_char(1485, '9,999')	' 1,485'	' 1,485'
to_char(1485, '9G999')	' 1,485'	' 1,485'
to_char(148.5, '999.999')	' 148.500'	' 148.500'
to_char(148.5, 'FM999.999')	'148.5'	'148.5'
to_char(148.5, 'FM999.990')	'148.500'	'148.500'
to_char(148.5, '999D999') -- com idioma	' 148,500'	' 148,500'
to_char(3148.5, '9G999D999')	' 3,148.500'	' 3,148.500'
to_char(-485, '999S')	'485-'	'485-'
to_char(-485, '999MI')	'485-'	'485-'
to_char(485, '999MI')	'485 '	'485 '
to_char(485, 'FM999MI')	'485'	'485'
to_char(485, 'PL999')	'+485'	formato inválido
to_char(485, 'SG999')	'+485'	formato inválido
to_char(-485, 'SG999')	'-485'	formato inválido
to_char(-485, '9SG99')	'4-85'	formato inválido
to_char(-485, '999PR')	'<485>'	'<485>'
to_char(485, 'L999') -- com idioma	'R\$ 485'	' R\$485'
to_char(485, 'RN')	' CDLXXXV'	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'	'CDLXXXV'
to_char(5.2, 'FMRN')	'V'	'V'
to_char(482, '999th')	' 482nd'	formato inválido
to_char(485, '"Good number:"999')	'Good number: 485'	formato inválido

Expressão	PostgreSQL 8.0.0 <sup>a</sup>	Oracle 10g (N. do T.) <sup>b</sup>
to_char(485.8, "Pre:"999" Post:" .999')	'Pre: 485 Post: .800'	formato inválido
to_char(12, '99V999')	' 12000'	' 12000'
to_char(12.4, '99V999')	' 12400'	' 12400'
to_char(12.45, '99V9')	' 125'	' 125'
Notas: a. idioma — set lc_numeric to 'pt_BR'; set lc_monetary to 'pt_BR'; (N. do T.) b. idioma — ALTER SESSION SET NLS_TERRITORY="BRAZIL"; (N. do T.)		

## 9.9. Funções e operadores para data e hora

A Tabela 9-28 mostra as funções disponíveis para processamento de valor de data e de hora. Os detalhes são mostrados nas próximas subseções. A Tabela 9-27 mostra o comportamento dos operadores aritméticos básicos (+, \*, etc.). Para as funções de formatação consulte a Seção 9.8. É necessário estar familiarizado com os tipos de dado para data e hora presentes na Seção 8.5.

Todas as funções e operadores descritos abaixo, que recebem os tipos time ou timestamp como entrada, estão presentes em duas formas: uma que recebe time with time zone ou timestamp with time zone, e outra que recebe time without time zone ou timestamp without time zone. Para abreviar, estas formas não são mostradas em separado. Também, os operadores + e \* ocorrem em pares comutativos (por exemplo, pares data + inteiro e inteiro + data); é mostrado apenas um destes pares.

**Tabela 9-27. Operadores para data e hora**

Operador	Exemplo	Resultado
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00'
+	time '01:00' + interval '3 hours'	time '04:00'
-	- interval '23 hours'	interval '-23:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3'
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00'
-	time '05:00' - time '03:00'	interval '02:00'
-	time '05:00' - interval '2 hours'	time '03:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00'
-	interval '1 day' - interval '1 hour'	interval '23:00'

Operador	Exemplo	Resultado
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00'
*	interval '1 hour' * double precision '3.5'	interval '03:30'
/	interval '1 hour' / double precision '1.5'	interval '00:40'

Tabela 9-28. Funções para data e hora

Função	Tipo retornado	Descrição	Exemplo	Resultado
age(timestamp, timestamp)	interval	Subtrai os argumentos, produzindo um resultado “simbólico” que utiliza anos e meses	age(timestamp '2001-04-10', timestamp '1957-06-13')	43 years 9 mons 27 days
age(timestamp)	interval	Subtrai de current_date	age(timestamp '1957-06-13')	43 years 8 mons 3 days
current_date	date	Data de hoje; consulte a Seção 9.9.4		
current_time	time with time zone	Hora do dia; consulte a Seção 9.9.4		
current_timestamp	timestamp with time zone	Data e hora; consulte a Seção 9.9.4		
date_part(text, timestamp)	double precision	Retorna subcampo (equivalente ao extract); consulte a Seção 9.9.1	date_part('hour', timestamp '2001-02-16 20:38:40')	20
date_part(text, interval)	double precision	Retorna subcampo (equivalente ao extract); consulte a Seção 9.9.1	date_part('month', interval '2 years 3 months')	3
date_trunc(text, timestamp)	timestamp	Trunca na precisão especificada; consulte a Seção 9.9.2	date_trunc('hour', timestamp '2001-02-16 20:38:40')	2001-02-16 20:00:00
extract(campo from timestamp)	double precision	Retorna subcampo; consulte a Seção 9.9.1	extract(hour from timestamp '2001-02-16 20:38:40')	20
extract(campo from interval)	double precision	Retorna subcampo; consulte a Seção 9.9.1	extract(month from interval '2 years 3 months')	3
isfinite(timestamp)	boolean	Testa carimbo do tempo finito (diferente de infinito)	isfinite(timestamp '2001-02-16 21:28:30')	true
isfinite(interval)	boolean	Testa intervalo finito	isfinite(interval)	true



Função	Tipo retornado	Descrição	Exemplo	Resultado
			1 '4 hours')	
localtime	time	Hora do dia; consulte a Seção 9.9.4		
localtimestamp	timestamp	Data e hora; consulte a Seção 9.9.4		
now()	timestamp with time zone	Data e hora corrente (equivalente ao <code>current_timestamp</code> ); consulte a Seção 9.9.4		
timeofday()	text	Data e hora corrente; consulte a Seção 9.9.4		

Além destas funções, é suportado o operador `OVERLAPS` do SQL:

```
( inicio1, fim1 ) OVERLAPS ( inicio2, fim2 )
( inicio1, duração1 ) OVERLAPS ( inicio2, duração2 )
```

O resultado desta expressão é verdade quando dois períodos de tempo (definidos por seus pontos limites) se sobrepõem, e falso quando não se sobrepõem. Os pontos limites podem ser especificados como pares de datas, horas, ou carimbo do tempo; ou como data, hora ou carimbo do tempo seguido por um intervalo.

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Resultado: verdade
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Resultado: falso
```

### 9.9.1. Funções `EXTRACT` e `date_part`

```
EXTRACT (campo FROM fonte)
```

A função `extract` retorna subcampos dos valores de data e hora, como o ano ou a hora. A *fonte* deve ser uma expressão de valor do tipo `timestamp`, `time` ou `interval` (As expressões do tipo `date` são convertidas em `timestamp`, podendo, portanto serem utilizadas também). O *campo* é um identificador, ou uma cadeia de caracteres, que seleciona o campo a ser extraído do valor fonte. A função `extract` retorna valores do tipo `double precision`. Abaixo estão mostrados os nomes de campo válidos:

`century`

O século

```
SELECT extract(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
Resultado: 20
SELECT extract(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 21
```

O primeiro século começou em 0001-01-01 00:00:00 DC, embora não se soubesse disso naquela época. Esta definição se aplica a todos os países que utilizam o calendário Gregoriano. Não existe o século número 0, vai direto de -1 para 1. Se você não concorda com isto, por favor envie sua reclamação para: Papa, Basílica de São Pedro, Cidade do Vaticano.

As versões do PostgreSQL anteriores a 8.0 não seguiam a numeração dos séculos convencional, retornando simplesmente o campo `ano` dividido por 100.

day

O campo dia (do mês) (1 - 31)

```
SELECT extract(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 16
```

decade

O campo ano dividido por 10

```
SELECT extract(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 200
```

dow

O dia da semana (0 - 6; Domingo é 0) (para valores timestamp apenas)

```
SELECT extract(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 5
```

Deve ser observado que a numeração do dia da semana da função extract (0 a 6) é diferente da numeração do dia da semana da função to\_char (1 a 7).

```
=> SELECT extract(DOW FROM TIMESTAMP '2005-08-14 20:38:40') AS Domingo;
```

```
domingo
-----
      0
(1 linha)
```

```
=> SELECT to_char(TIMESTAMP '2005-08-14 20:38:40','D') AS Domingo;
```

```
domingo
-----
      1
(1 linha)
```

```
=> SELECT extract(DOW FROM TIMESTAMP '2005-08-20 20:38:40') AS Sábado;
```

```
sábado
-----
      6
(1 linha)
```

```
=> SELECT to_char(TIMESTAMP '2005-08-20 20:38:40','D') AS Sábado;
```

```
sábado
-----
      7
(1 linha)
```

doy

O dia do ano (1 - 365/366) (para valores timestamp apenas)

```
SELECT extract(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Resultado: 47
```

Deve ser observado que a numeração do dia do ano da função extract (1 a 366) é igual a numeração do dia do ano da função to\_char (1 a 366).

```
=> SELECT extract(DOY FROM TIMESTAMP '2004-12-31 23:59:59') AS dia;
```

```
dia
-----
   366
(1 linha)
```

```
=> SELECT to_char(TIMESTAMP '2004-12-31 23:59:59', 'DDD') AS dia;
```

```
dia
-----
366
(1 linha)
```

#### epoch

Para valores `date` e `timestamp`, o número de segundos desde 1970-01-01 00:00:00-00 (pode ser negativo); para valores `interval`, o número total de segundos do intervalo

```
SELECT extract(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-08');
```

Resultado: 982384720

```
SELECT extract(EPOCH FROM INTERVAL '5 days 3 hours');
```

Resultado: 442800

Abaixo está mostrado como converter de volta um valor de época para um valor de carimbo do tempo:

```
SELECT TIMESTAMP WITH TIME ZONE 'epoch' + 982384720 * INTERVAL '1 second';
```

#### hour

O campo hora (0 - 23)

```
SELECT extract(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
```

Resultado: 20

#### microseconds

O campo segundos, incluindo a parte fracionária, multiplicado por 1 milhão (1.000.000). Deve ser observado que inclui os segundos decorridos, e não apenas a fração de segundos.

```
SELECT extract(MICROSECONDS FROM TIME '17:12:28.5');
```

Resultado: 28500000

#### millennium

O milênio

```
SELECT extract(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
```

Resultado: 3

Os anos em 1900 estão no segundo milênio. O terceiro milênio começou em 1 de janeiro de 2001.

As versões do PostgreSQL anteriores a 8.0 não seguiam a numeração dos milênios convencional, retornando simplesmente o campo ano dividido por 1000.

#### milliseconds

O campo segundos, incluindo a parte fracionária, multiplicado por mil (1.000). Deve ser observado que inclui os segundos decorridos, e não apenas a fração de segundos.

```
SELECT extract(MILLISECONDS FROM TIME '17:12:28.5');
```

Resultado: 28500

#### minute

O campo minutos (0 - 59)

```
SELECT extract(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
```

Resultado: 38

#### month

Para valores `timestamp`, o número do mês do ano dentro do ano (1 - 12); para valores `interval`, o número de meses, módulo 12 (0 - 11)

```
SELECT extract(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
```

Resultado: 2

```
SELECT extract(MONTH FROM INTERVAL '2 years 3 months');
```

Resultado: 3

```
SELECT extract(MONTH FROM INTERVAL '2 years 13 months');
```

Resultado: 1

quarter

O trimestre do ano (1 - 4) onde o dia se encontra (para valores `timestamp` apenas)

```
SELECT extract(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
```

Resultado: 1

second

O campo segundos, incluindo a parte fracionária (0 - 59)<sup>9 10</sup>

```
SELECT extract(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
```

Resultado: 40

```
SELECT extract(SECOND FROM TIME '17:12:28.5');
```

Resultado: 28.5

timezone

O deslocamento da zona horária em relação à UTC, medido em segundos. Os valores positivos correspondem às zonas horárias a leste da UTC, e os valores negativos correspondem às zonas horárias a oeste da UTC.<sup>11</sup>

timezone\_hour

O componente hora do deslocamento da zona horária

timezone\_minute

O componente minuto do deslocamento da zona horária

week

O número da semana do ano onde o dia se encontra. Por definição (ISO 8601), a primeira semana do ano contém o dia 4 de janeiro deste ano; a semana ISO-8601 começa na segunda-feira. Em outras palavras, a primeira quinta-feira do ano está na primeira semana do ano. (apenas para valores `timestamp`)

```
SELECT extract(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
```

Resultado: 7

year

O campo ano. Deve-se ter em mente que não existe o ano 0 DC e, portanto, subtrair anos AC de DC deve ser feito com cautela.

```
SELECT extract(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
```

Resultado: 2001

A função `extract` é voltada principalmente para o processamento computacional. Para formatar valores de data e hora para exibição, consulte a Seção 9.8.

A função `date_part` é modelada segundo a função equivalente tradicional do Ingres à função `extract` do padrão SQL:

```
date_part('campo', fonte)
```

Deve ser observado que, neste caso, o parâmetro `campo` deve ser um valor cadeia de caracteres, e não um nome. Os nomes de `campo` válidos para `date_part` são os mesmos da função `extract`.

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
```

Resultado: 16

```
SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
```

Resultado: 4

### 9.9.2. date\_trunc

A função `date_trunc` é conceitualmente similar à função `trunc` para números.

```
date_trunc('campo', fonte)
```

*fonte* é uma expressão de valor do tipo `timestamp` ou `interval` (valores do tipo `date` e `time` são convertidos automaticamente em `timestamp` ou `interval`, respectivamente). O *campo* seleciona a precisão a ser utilizada para truncar o valor da entrada. O valor retornado é do tipo `timestamp` ou `interval`, com todos os campos menos significativos do que valor selecionado tornados zero (ou um, para o dia do mês).

Os valores válidos para *campo* são:

```
microseconds
milliseconds
second
minute
hour
day
week
month
year
decade
century
millennium
```

Exemplos:

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
```

Resultado: 2001-02-16 20:00:00

```
SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
```

Resultado: 2001-01-01 00:00:00

### 9.9.3. AT TIME ZONE

A construção `AT TIME ZONE` permite a conversão do carimbo do tempo para uma zona horária diferente. A Tabela 9-29 mostra suas variantes.

**Tabela 9-29. Variantes de AT TIME ZONE**

Expressão	Tipo retornado	Descrição
<code>timestamp without time zone AT TIME ZONE zona</code>	<code>timestamp with time zone</code>	Converte hora local de uma determinada zona horária para UTC
<code>timestamp with time zone AT TIME ZONE zona</code>	<code>timestamp without time zone</code>	Converte de UTC para a hora local em uma determinada zona horária
<code>time with time zone AT TIME ZONE zona</code>	<code>time with time zone</code>	Converte hora local entre zonas horárias

Nestas expressões, a *zona* da zona horária desejada, pode ser especificada tanto por meio de um texto em uma cadeia de caracteres (por exemplo, `'PST'`), quanto por um intervalo (por exemplo, `INTERVAL '-08:00'`). No caso do texto, os nomes disponíveis para zona horária são os mostrados na Tabela B-4 (seria mais útil suportar os nomes mais gerais mostrados na Tabela B-6, mas isto ainda não está implementado).

Exemplos (supondo que a zona horária local seja `PST8PDT`):

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'MST';
```

Resultado: 2001-02-16 19:38:40-08

```
SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'MST';
```

Resultado: 2001-02-16 18:38:40

O primeiro exemplo recebe um carimbo do tempo sem zona horária e o interpreta como hora MST (UTC-7) para produzir um carimbo do tempo UTC, o qual é então rotacionado para PST (UTC-8) para ser exibido. O segundo exemplo recebe um carimbo do tempo especificado em EST (UTC-5) e converte para hora local MST (UTC-7).

A função `timezone(zona, carimbo_do_tempo)` equivale à construção em conformidade com o padrão SQL `carimbo_do_tempo AT TIME ZONE zona`.

#### 9.9.4. Data e hora corrente

Estão disponíveis as seguintes funções para obter a data e hora corrente:

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME ( precisão )
CURRENT_TIMESTAMP ( precisão )
LOCALTIME
LOCALTIMESTAMP
LOCALTIME ( precisão )
LOCALTIMESTAMP ( precisão )
```

`CURRENT_TIME` e `CURRENT_TIMESTAMP` retornam valores com zona horária; `LOCALTIME` e `LOCALTIMESTAMP` retornam valores sem zona horária.

`CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME` e `LOCALTIMESTAMP` podem, opcionalmente, receber o parâmetro `precisão` fazendo o resultado ser arredondado nesta quantidade de dígitos fracionários no campo de segundos. Sem o parâmetro de `precisão`, o resultado é produzido com toda a precisão disponível.

**Nota:** Antes do PostgreSQL versão 7.2, os parâmetros de `precisão` não estavam implementados, e o resultado era sempre retornado em segundos inteiros.

Alguns exemplos:

```
SELECT CURRENT_TIME;
Resultado: 14:39:53.662522-05
```

```
SELECT CURRENT_DATE;
Resultado: 2001-12-23
```

```
SELECT CURRENT_TIMESTAMP;
Resultado: 2001-12-23 14:39:53.662522-05
```

```
SELECT CURRENT_TIMESTAMP(2);
Resultado: 2001-12-23 14:39:53.66-05
```

```
SELECT LOCALTIMESTAMP;
Resultado: 2001-12-23 14:39:53.662522
```

A função `now()` é o equivalente tradicional do PostgreSQL para `CURRENT_TIMESTAMP`.

Também existe a função `timeofday()`, que por motivos históricos retorna uma cadeia de caracteres do tipo `text`, e não um valor do tipo `timestamp`:

```
SELECT timeofday();
Resultado: Sat Feb 17 19:07:32.000126 2001 EST
```

É importante saber que `CURRENT_TIMESTAMP`, e as funções relacionadas, retornam a data e hora do começo da transação corrente; seus valores não mudam durante a transação. Isto é considerado uma funcionalidade: o objetivo é permitir que a transação possua uma noção consistente do tempo “corrente”, de forma que várias modificações dentro da mesma transação compartilhem o mesmo carimbo do tempo. A função `timeofday()` retorna a hora do relógio, avançando durante as transações.

**Nota:** Outros sistemas de banco de dados podem avançar estes valores com mais frequência.

Todos os tipos de dado para data e hora também aceitam o valor literal especial `now` para especificar a data e hora corrente. Portanto, os três comandos abaixo retornam o mesmo resultado:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now'; -- incorreto para uso com DEFAULT
```

**Dica:** Não se utiliza a terceira forma ao especificar a cláusula `DEFAULT` na criação da tabela. O sistema converte `now` em `timestamp` tão logo a constante é analisada e, portanto, quando o valor padrão for utilizado será utilizada a hora da criação da tabela! As duas primeiras formas não são avaliadas até que o valor padrão seja utilizado, porque são chamadas de função. Assim sendo, as duas primeiras formas fornecem o comportamento desejado quando o padrão for a hora de inserção da linha.

### 9.9.5. Comparação entre o PostgreSQL, o Oracle, o SQL Server e o DB2

**Nota:** Seção escrita pelo tradutor, não fazendo parte do manual original.

Esta seção tem por finalidade comparar, através de exemplos práticos, as funções e operadores para data e hora do PostgreSQL, do Oracle, do SQL Server e do DB2.

#### Exemplo 9-9. Utilização de INTERVAL

Abaixo estão mostrados exemplos comparando a utilização de `INTERVAL` no PostgreSQL e no Oracle. Consulte também Interval Literals ([http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/sql\\_elements3a.htm#38599](http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/sql_elements3a.htm#38599)).

PostgreSQL 8.0.0:

```
=> SELECT INTERVAL'20 DAY' - INTERVAL'240 HOUR' AS intervalo;
```

```
intervalo
-----
10 days
(1 linha)
```

```
=> SELECT INTERVAL '4 DAYS 5 HOURS 12 MINUTES';
```

```
interval
-----
4 days 05:12:00
(1 linha)
```

Oracle 10g:

```
SQL> SELECT INTERVAL'20' DAY - INTERVAL'240' HOUR(3) FROM sys.dual;
```

```
INTERVAL'20'DAY-INTERVAL'240'HOUR(3)
-----
+0000000010 00:00:00.000000000
```

```
SQL> SELECT INTERVAL '4 5:12' DAY TO MINUTE FROM sys.dual;
```

```
INTERVAL'45:12'DAYTOMINUTE
-----
+04 05:12:00
```

#### Exemplo 9-10. Número de dias entre duas datas

Abaixo estão mostrados exemplos de funções do PostgreSQL, do SQL Server, do Oracle e do DB2, para obter o número de dias entre duas datas. Consulte também Getting the difference between Dates (<http://asktom.oracle.com/~tkyte/Misc/DateDiff.html>), DB2 Basics: Fun with Dates and Times (<http://www-128.ibm.com/developerworks/db2/library/techarticle/0211yip/0211yip3.html>) e SQL Server, DATEDIFF ([http://msdn.microsoft.com/library/en-us/tsqlref/ts\\_da-db\\_5vxi.asp?frame=true](http://msdn.microsoft.com/library/en-us/tsqlref/ts_da-db_5vxi.asp?frame=true))

PostgreSQL 8.0.0:

```
=> SELECT date('1950-07-16') - date('1949-11-21') AS dias;
```

```

dias
-----
    237
(1 linha)

```

SQL Server 2000:

```
SELECT    datediff(DAY,      convert(datetime,'1949-11-21',120),      convert(datetime,'1950-07-16',120)) AS dias
```

```

dias
----
    237
(1 row(s) affected)

```

Oracle 10g:

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD';
```

```
SQL> SELECT to_date('1950-07-16') - to_date('1949-11-21') AS dias FROM sys.dual;
```

```

      DIAS
-----
      237

```

DB2 8.1:

```
DB2SQL92> SELECT    days(date('1950-07-16'))-    days(date('1949-11-21'))    AS    dias    FROM
sysibm.sysdummy1;
```

```

DIAS
-----
    237

```

### Exemplo 9-11. Obtenção do dia do mês

Abaixo estão mostrados exemplos de funções do PostgreSQL, do SQL Server, do Oracle e do DB2, para obter o dia do mês. Consulte também TO\_CHAR (datetime) no Oracle (<http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/functions134a.htm>), DAY no DB2 (<https://aurora.vcu.edu/db2help/db2s0/qls0414.htm#HDRFNDAY>) e SQL Server Date and Time Functions ([http://msdn.microsoft.com/library/en-us/tsqlref/ts\\_fa-fz\\_2c1f.asp](http://msdn.microsoft.com/library/en-us/tsqlref/ts_fa-fz_2c1f.asp))

PostgreSQL 8.0.0:

```
=> SELECT date_part('day',CURRENT_TIMESTAMP) AS dia;
```

```

dia
-----
   14
(1 linha)

```

```
=> SELECT extract(DAY FROM CURRENT_TIMESTAMP) AS dia;
```

```

dia
-----
   14
(1 linha)

```

```
=> SELECT to_char(CURRENT_TIMESTAMP,'DD') AS dia;
```

```

dia
-----
   14
(1 linha)

```



SQL Server 2000:

```
SELECT datepart(DAY, CURRENT_TIMESTAMP) AS dia
```

```
dia
---
14
(1 row(s) affected)
```

```
SELECT day(CURRENT_TIMESTAMP) AS dia
```

```
dia
---
14
(1 row(s) affected)
```

Oracle 10g:

```
SQL> SELECT to_char(CURRENT_TIMESTAMP,'DD') AS dia FROM sys.dual;
```

```
DI
--
14
```

```
SQL> SELECT extract(DAY FROM CURRENT_TIMESTAMP) AS dia FROM sys.dual;
```

```
DI
--
14
```

DB2 8.1:

```
DB2SQL92> SELECT day(CURRENT_TIMESTAMP) FROM sysibm.sysdummy1;
```

```
1
-----
14
```

### Exemplo 9-12. Utilização das funções para data e hora corrente

Abaixo estão mostrados exemplos comparando a utilização das funções que retornam a data e hora corrente no PostgreSQL, no SQL Server, no Oracle e no DB2.<sup>12</sup> Consulte também CURRENT\_TIMESTAMP no Oracle (<http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/functions31a.htm>), CURRENT\_TIMESTAMP no SQL Server ([http://msdn.microsoft.com/library/en-us/tsqlref/ts\\_cr-cz\\_60mo.asp?frame=true](http://msdn.microsoft.com/library/en-us/tsqlref/ts_cr-cz_60mo.asp?frame=true)) e CURRENT\_TIMESTAMP no DB2 (<https://aurora.vcu.edu/db2help/db2s0/qls031j.htm#HDC2CURTS>).

Como o PostgreSQL, o SQL Server e o Oracle foram executados no Windows 2000, enquanto o DB2 foi executado no Linux, deve ser observado que a precisão da fração de segundos varia.

PostgreSQL 8.0.0:

```
=> SELECT CURRENT_DATE;
```

```
date
-----
2005-03-04
(1 linha)
```

```
=> SELECT CURRENT_TIME;
```

```
timetz
-----
16:42:07.33-03
(1 linha)
```

```
=> SELECT CURRENT_TIMESTAMP;
```

```

            timestampz
-----
2005-03-04 16:42:07.38-03
(1 linha)

=> SELECT CURRENT_TIME(0);

            timetz
-----
16:42:07-03
(1 linha)

=> SELECT CURRENT_TIMESTAMP(0);

            timestampz
-----
2005-03-04 16:42:07-03
(1 linha)

=> SELECT LOCALTIME AS agora;

            agora
-----
16:42:07.53
(1 linha)

=> SELECT LOCALTIMESTAMP;

            timestamp
-----
2005-03-04 16:42:07.58
(1 linha)

=> SELECT LOCALTIME (0) AS agora;

            agora
-----
16:42:08
(1 linha)

=> SELECT LOCALTIMESTAMP (0) AS agora;

            agora
-----
2005-03-04 16:42:08
(1 linha)

=> SELECT CURRENT_TIMESTAMP AT TIME ZONE 'UTC' AS UTC;

            utc
-----
2005-03-04 19:42:07.73
(1 linha)

=> SELECT now();

            now
-----
2005-03-04 16:42:07.891-03
(1 linha)

```

```
=> SELECT timeofday();
```

```

          timeofday
-----
Fri Mar 04 16:42:07.941000 2005 BRT
(1 linha)
```

```
=> SELECT to_char(CURRENT_TIMESTAMP, 'DD-MM-YYYY HH24:MI:SS') AS agora;
```

```

          agora
-----
04-03-2005 16:42:07
(1 linha)
```

SQL Server 2000:

```
SELECT CURRENT_TIMESTAMP AS agora
```

```

agora
-----
2005-03-04 16:39:51.207
(1 row(s) affected)
```

```
SELECT getdate() AS agora
```

```

agora
-----
2005-03-04 16:39:51.207
(1 row(s) affected)
```

```
SELECT getutcdate() AS UTC
```

```

UTC
-----
2005-03-04 19:39:51.207
(1 row(s) affected)
```

```
SELECT convert(VARCHAR, CURRENT_TIMESTAMP, 121) AS agora
```

```

agora
-----
2005-03-04 16:39:51.207
(1 row(s) affected)
```

Oracle 10g:

```
SQL> ALTER SESSION SET TIME_ZONE = local;
```

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD';
```

```
SQL> ALTER SESSION SET NLS_TIMESTAMP_FORMAT = 'YYYY-MM-DD HH24:MI:SS.FF';
```

```
SQL> ALTER SESSION SET NLS_TIMESTAMP_TZ_FORMAT = 'YYYY-MM-DD HH24:MI:SS.FF TZh:TzM';
```

```
SQL> SELECT CURRENT_DATE FROM sys.dual;
```

```

CURRENT_DA
-----
2005-03-26
```

```
SQL> SELECT CURRENT_TIMESTAMP FROM sys.dual;
```

```

CURRENT_TIMESTAMP
-----
2005-03-26 17:31:33.934000 -03:00
```

```
SQL> SELECT CURRENT_TIMESTAMP(0) FROM sys.dual;
```

```
CURRENT_TIMESTAMP(0)
-----
2005-03-26 17:31:34. -03:00
```

```
SQL> SELECT LOCALTIMESTAMP FROM sys.dual;
```

```
LOCALTIMESTAMP
-----
2005-03-26 17:31:33.954000
```

```
SQL> SELECT LOCALTIMESTAMP (0) FROM sys.dual;
```

```
LOCALTIMESTAMP(0)
-----
2005-03-26 17:31:34.
```

```
SQL> SELECT CURRENT_TIMESTAMP AT TIME ZONE 'UTC' AS UTC FROM sys.dual;
```

```
UTC
-----
2005-03-26 20:31:33.984000 +00:00
```

```
SQL> SELECT to_char(CURRENT_TIMESTAMP, 'DD-MM-YYYY HH24:MI:SS') AS agora FROM sys.dual;
```

```
AGORA
-----
26-03-2005 17:31:33
```

DB2 8.1:

```
DB2SQL92> SELECT CURRENT_DATE AS data FROM sysibm.sysdummy1;
```

```
DATA
-----
26/03/2005
```

```
DB2SQL92> SELECT CURRENT DATE AS data FROM sysibm.sysdummy1;
```

```
DATA
-----
26/03/2005
```

```
DB2SQL92> SELECT CURRENT_TIME AS hora FROM sysibm.sysdummy1;
```

```
HORA
-----
21:55:05
```

```
DB2SQL92> SELECT CURRENT TIME AS hora FROM sysibm.sysdummy1;
```

```
HORA
-----
21:55:25
```

```
DB2SQL92> SELECT CURRENT_TIMESTAMP AS agora FROM sysibm.sysdummy1;
```

```
AGORA
-----
2005-03-26-21.55.47.195177
```

```
DB2SQL92> SELECT CURRENT TIMESTAMP AS agora FROM sysibm.sysdummy1;

AGORA
-----
2005-03-26-21.56.13.294096

DB2SQL92> SELECT CURRENT TIMESTAMP - MICROSECOND (CURRENT TIMESTAMP) MICROSECONDS
DB2SQL92>          AS "CURRENT_TIMESTAMP(0)" FROM sysibm.sysdummy1;

CURRENT_TIMESTAMP(0)
-----
2005-03-26-21.56.42.000000

DB2SQL92> SELECT CURRENT TIMESTAMP - CURRENT TIMEZONE AS utc FROM sysibm.sysdummy1;

UTC
-----
2005-03-27-00.57.19.704003
```

### Exemplo 9-13. Tipo de dado timestamp

Abaixo são mostrados exemplos comparando a utilização do tipo de dado timestamp no PostgreSQL, no SQL Server, no Oracle e no DB2.

PostgreSQL 8.0.0:

```
=> SELECT TIMESTAMP '1999-01-01 12:34:56' AS timestamp;

      timestamp
-----
1999-01-01 12:34:56
(1 linha)

=> SELECT cast('1999-01-01 12:34:56' AS TIMESTAMP) AS timestamp;

      timestamp
-----
1999-01-01 12:34:56
(1 linha)
```

SQL Server 2000:

```
SELECT cast('1999-01-01 12:34:56' AS datetime) AS timestamp

timestamp
-----
1999-01-01 12:34:56.000
(1 row(s) affected)

SELECT convert(datetime,'1999-01-01 12:34:56',120) AS timestamp

timestamp
-----
1999-01-01 12:34:56.000
(1 row(s) affected)
```

Oracle 10g:

```
SQL> ALTER SESSION SET TIME_ZONE = local;
SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD';
SQL> ALTER SESSION SET NLS_TIMESTAMP_FORMAT = 'YYYY-MM-DD HH24:MI:SS.FF';
SQL> ALTER SESSION SET NLS_TIMESTAMP_TZ_FORMAT = 'YYYY-MM-DD HH24:MI:SS.FF TZH:TZM';

SQL> SELECT TIMESTAMP '1999-01-01 12:34:56' AS timestamp FROM sys.dual;
```

```

TIMESTAMP
-----
1999-01-01 12:34:56.000000000

```

```
SQL> SELECT cast('1999-01-01 12:34:56' AS TIMESTAMP) AS timestamp FROM sys.dual;
```

```

TIMESTAMP
-----
1999-01-01 12:34:56.0000000

```

DB2 8.1:

```
DB2SQL92> SELECT timestamp('1999-01-01','12.34.56') AS timestamp FROM sysibm.sysdummy1;
```

```

TIMESTAMP
-----
1999-01-01-12.34.56.0000000

```

```
DB2SQL92> SELECT cast('1999-01-01 12:34:56' AS TIMESTAMP) AS timestamp FROM sysibm.sysdummy1;
```

```

TIMESTAMP
-----
1999-01-01-12.34.56.0000000

```

#### Exemplo 9-14. Somar dias e horas a uma data

Abaixo são mostrados exemplos de aritmética com datas, somando 30 dias a uma data sem hora, 30 dias a uma data com hora, e 3 dias e 3 horas ao carimbo do tempo corrente, no PostgreSQL, no Oracle, no SQL Server e no DB2.<sup>13</sup>

PostgreSQL 8.0.0:

```
=> SELECT cast('2004-07-16' AS date) + interval'30 days' AS data;
```

```

      data
-----
2004-08-15 00:00:00
(1 linha)

```

```
=> SELECT cast('2004-07-16 15:00:00' AS timestamp) + interval'30 days' AS data;
```

```

      data
-----
2004-08-15 15:00:00
(1 linha)

```

```
=> SELECT current_timestamp as agora,
-> current_timestamp + interval'3 days 3 hours' AS depois;
```

```

      agora |      depois
-----+-----
2005-04-01 08:16:44.029386-03 | 2005-04-04 11:16:44.029386-03
(1 linha)

```

SQL Server 2000:

```
SELECT dateadd(DAY,30,convert(smallerdatetime,'2004-07-16',120)) as data
```

```

data
-----
2004-08-15 00:00:00
(1 row(s) affected)

```

```
SELECT dateadd(DAY,30,convert(datetime,'2004-07-16 15:00:00',120)) as data
```

```
data
```

```
-----
```

```
2004-08-15 15:00:00.000
```

```
(1 row(s) affected)
```

```
SELECT current_timestamp as agora,
       dateadd(DAY,3,dateadd(HOUR,3,current_timestamp)) as depois
```

```
agora
```

```
depois
```

```
-----
```

```
2005-04-01 06:27:41.367 2005-04-04 09:27:41.367
```

```
(1 row(s) affected)
```

Oracle 10g:

```
SQL> ALTER SESSION SET TIME_ZONE = local;
```

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH24:MI:SS';
```

```
SQL> ALTER SESSION SET NLS_TIMESTAMP_FORMAT = 'YYYY-MM-DD HH24:MI:SS.FF';
```

```
SQL> ALTER SESSION SET NLS_TIMESTAMP_TZ_FORMAT = 'YYYY-MM-DD HH24:MI:SS.FF TZh:TzM';
```

```
SQL> SELECT cast('2004-07-16' AS date) + 30
       2 FROM sys.dual;
```

```
CAST('2004-07-16' AS
```

```
-----
```

```
2004-08-15 00:00:00
```

```
SQL> SELECT cast('2004-07-16' AS date) + INTERVAL '30' DAY
       2 FROM sys.dual;
```

```
CAST('2004-07-16' AS
```

```
-----
```

```
2004-08-15 00:00:00
```

```
SQL> SELECT cast('2004-07-16 15:00:00' AS timestamp) + 30
       2 FROM sys.dual;
```

```
CAST('2004-07-1615:
```

```
-----
```

```
2004-08-15 15:00:00
```

```
SQL> SELECT cast('2004-07-16 15:00:00' AS timestamp) + INTERVAL '30' DAY
       2 FROM sys.dual;
```

```
CAST('2004-07-1615:00:00' AS TIMESTAMP) + INTERVAL '30' DAY
```

```
-----
```

```
2004-08-15 15:00:00.000000000
```

```
SQL> SELECT current_timestamp AS agora, current_timestamp + 3 + 3/24 AS depois
       2 FROM sys.dual;
```

```
AGORA
```

```
DEPOIS
```

```
-----
```

```
2005-04-01 18:04:58.702000 -03:00 2005-04-04 21:04:58
```

```
SQL> SELECT current_timestamp AS agora,
       2 current_timestamp + interval '3' day + interval '3' hour AS depois
       3 FROM sys.dual;
```

```
AGORA
```

```
DEPOIS
```

```
-----
```

```
2005-04-01 18:04:59.253000 -03:00 2005-04-04 21:04:59.253000000 -03:00
```

DB2 8.1:

```
DB2SQL92> SELECT cast('2004-07-16' AS date) + 30 days
DB2SQL92> FROM sysibm.sysdummy1;
```

```
1
-----
15/08/2004
```

```
DB2SQL92> SELECT cast('2004-07-16 15:00:00' AS timestamp) + 30 days
DB2SQL92> FROM sysibm.sysdummy1;
```

```
1
-----
2004-08-15-15.00.00.000000
```

```
DB2SQL92> SELECT current_timestamp, current_timestamp + 3 days + 3 hours
DB2SQL92> FROM sysibm.sysdummy1;
```

```
1                                2
-----
2005-04-01-07.34.41.953274    2005-04-04-10.34.41.953274
```

## 9.10. Funções e operadores geométricos

Os tipos geométricos point, box, lseg, line, path, polygon e circle possuem um amplo conjunto de funções e operadores nativos para apoiá-los, mostrados na Tabela 9-30, na Tabela 9-31 e na Tabela 9-32.

**Tabela 9-30. Operadores geométricos**

Operador	Descrição	Exemplo
+	Translação	box '((0,0),(1,1))' + point '(2.0,0)'
-	Translação	box '((0,0),(1,1))' - point '(2.0,0)'
*	Escala/rotação	box '((0,0),(1,1))' * point '(2.0,0)'
/	Escala/rotação	box '((0,0),(2,2))' / point '(2.0,0)'
#	Ponto ou caixa de interseção	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	Número de pontos do caminho ou do polígono	# '((1,0),(0,1),(-1,0))'
@-@	Comprimento ou circunferência	@-@ path '((0,0),(1,0))'
@@	Centro	@@ circle '((0,0),10)'
##	Ponto mais próximo do primeiro operando no segundo operando	point '(0,0)' ## lseg '((2,0),(0,2))'
<->	Distância entre	circle '((0,0),1)' <-> circle '((5,0),1)'
&&	Se sobrepõem?	box '((0,0),(1,1))' && box '((0,0),(2,2))'
&<	Não se estende à direita de?	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	Não se estende à esquerda de?	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<<	Está à esquerda?	circle '((0,0),1)' << circle '((5,0),1)'



Operador	Descrição	Exemplo
>>	Está à direita?	<code>circle '((5,0),1)' &gt;&gt; circle '((0,0),1)'</code>
<^	Está abaixo?	<code>circle '((0,0),1)' &lt;^ circle '((0,5),1)'</code>
>^	Está acima?	<code>circle '((0,5),1)' &gt;^ circle '((0,0),1)'</code>
?#	Se intersectam?	<code>lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'</code>
?-	É horizontal?	<code>?- lseg '((-1,0),(1,0))'</code>
?-	São alinhados horizontalmente?	<code>point '(1,0)' ?- point '(0,0)'</code>
?	É vertical?	<code>?  lseg '((-1,0),(1,0))'</code>
?	São alinhados verticalmente	<code>point '(0,1)' ?  point '(0,0)'</code>
?-	São perpendiculares?	<code>lseg '((0,0),(0,1))' ?-  lseg '((0,0),(1,0))'</code>
?	São paralelos?	<code>lseg '((-1,0),(1,0))' ?   lseg '((-1,2),(1,2))'</code>
~	Contém?	<code>circle '((0,0),2)' ~ point '(1,1)'</code>
@	Está contido ou sobre?	<code>point '(1,1)' @ circle '((0,0),2)'</code>
~=	O mesmo que?	<code>polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'</code>

Tabela 9-31. Funções geométricas

Função	Tipo retornado	Descrição	Exemplo
<code>area(object)</code>	double precision	área	<code>area(box '((0,0),(1,1))')</code>
<code>box_intersect(box, box)</code>	box	caixa de interseção	<code>box_intersect(box '((0,0),(1,1))', box '((0.5,0.5),(2,2))')</code>
<code>center(object)</code>	point	centro	<code>center(box '((0,0),(1,2))')</code>
<code>diameter(circle)</code>	double precision	diâmetro do círculo	<code>diameter(circle '((0,0),2.0)')</code>
<code>height(box)</code>	double precision	tamanho vertical da caixa	<code>height(box '((0,0),(1,1))')</code>
<code>isclosed(path)</code>	boolean	é um caminho fechado?	<code>isclosed(path '((0,0),(1,1),(2,0))')</code>
<code>isopen(path)</code>	boolean	é um caminho aberto?	<code>isopen(path '[(0,0),(1,1),(2,0)]')</code>
<code>length(object)</code>	double precision	comprimento	<code>length(path '((-1,0),(1,0))')</code>
<code>npoints(path)</code>	integer	número de pontos	<code>npoints(path '[(0,0),(1,1),(2,0)]')</code>
<code>npoints(polygon)</code>	integer	número de pontos	<code>npoints(polygon '((1,1),(0,0))')</code>

Função	Tipo retornado	Descrição	Exemplo
<code>pclose(path)</code>	<code>path</code>	converte o caminho em caminho fechado	<code>pclose(path '[(0,0),(1,1),(2,0)]')</code>
<code>popen(path)</code>	<code>path</code>	converte o caminho em caminho aberto	<code>popen(path '((0,0),(1,1),(2,0))')</code>
<code>radius(circle)</code>	<code>double precision</code>	raio do círculo	<code>radius(circle '((0,0),2.0)')</code>
<code>width(box)</code>	<code>double precision</code>	tamanho horizontal da caixa	<code>width(box '((0,0),(1,1))')</code>

Tabela 9-32. Funções de conversão de tipo geométrico

Função	Tipo retornado	Descrição	Exemplo
<code>box(circle)</code>	<code>box</code>	círculo em caixa	<code>box(circle '((0,0),2.0)')</code>
<code>box(point, point)</code>	<code>box</code>	pontos em caixa	<code>box(point '(0,0)', point '(1,1)')</code>
<code>box(polygon)</code>	<code>box</code>	polígono em caixa	<code>box(polygon '((0,0),(1,1),(2,0))')</code>
<code>circle(box)</code>	<code>circle</code>	caixa em círculo	<code>circle(box '((0,0),(1,1))')</code>
<code>circle(point, double precision)</code>	<code>circle</code>	centro e raio em círculo	<code>circle(point '(0,0)', 2.0)</code>
<code>lseg(box)</code>	<code>lseg</code>	diagonal de caixa em segmento de linha	<code>lseg(box '((-1,0),(1,0))')</code>
<code>lseg(point, point)</code>	<code>lseg</code>	ponto em segmento de linha	<code>lseg(point '(-1,0)', point '(1,0)')</code>
<code>path(polygon)</code>	<code>point</code>	polígono em caminho	<code>path(polygon '((0,0),(1,1),(2,0))')</code>
<code>point(double precision, double precision)</code>	<code>point</code>	constrói ponto	<code>point(23.4, -44.5)</code>
<code>point(box)</code>	<code>point</code>	centro da caixa	<code>point(box '((-1,0),(1,0))')</code>
<code>point(circle)</code>	<code>point</code>	centro do círculo	<code>point(circle '((0,0),2.0)')</code>
<code>point(lseg)</code>	<code>point</code>	centro do segmento de linha	<code>point(lseg '((-1,0),(1,0))')</code>
<code>point(lseg, lseg)</code>	<code>point</code>	intersecção	<code>point(lseg '((-1,0),(1,0))', lseg '((-2,-2),(2,2))')</code>
<code>point(polygon)</code>	<code>point</code>	centro do polígono	<code>point(polygon '((0,0),(1,1),(2,0))')</code>
<code>polygon(box)</code>	<code>polygon</code>	caixa em polígono de 4 pontos	<code>polygon(box '((0,0),(1,1))')</code>
<code>polygon(circle)</code>	<code>polygon</code>	círculo em polígono de 12 pontos	<code>polygon(circle '((0,0),2.0)')</code>
<code>polygon(npts, circle)</code>	<code>polygon</code>	círculo em polígono de <i>npts</i> -pontos	<code>polygon(12, circle '((0,0),2.0)')</code>

Função	Tipo retornado	Descrição	Exemplo
<code>polygon(path)</code>	<code>polygon</code>	caminho em polígono	<code>polygon(path '((0,0),(1,1),(2,0))')</code>

É possível acessar os dois números que compõem um `point` como se este fosse uma matriz com os índices 0 e 1. Por exemplo, se `t.p` for uma coluna do tipo `point`, então `SELECT p[0] FROM t` retorna a coordenada X, e `UPDATE t SET p[1] = ...` altera a coordenada Y. Do mesmo modo, um valor do tipo `box` ou `lseg` pode ser tratado como sendo uma matriz contendo dois valores do tipo `point`.

As funções `area` operam sobre os tipos `box`, `circle` e `path`. A função `area` somente opera sobre o tipo de dado `path` se os pontos em `path` não se intersectarem. Por exemplo, não opera sobre o `path` `'((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))':PATH`, entretanto opera sobre o `path` visualmente idêntico `'((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))':PATH`. Se o conceito de `path` que intersecta e que não intersecta estiver confuso, desenhe os dois caminhos acima lado a lado em uma folha de papel gráfico.

## 9.11. Funções e operadores para endereço de rede

A Tabela 9-33 mostra os operadores disponíveis para uso com os tipos `cidr` e `inet`. Os operadores `<`, `<=`, `>` e `>=` testam a inclusão na subrede: somente consideram a parte de rede dos dois endereços, ignorando qualquer parte de hospedeiro, determinando se a parte de rede é idêntica, ou se uma é subrede da outra.

Tabela 9-33. Operadores para os tipos `cidr` e `inet`

Operador	Descrição	Exemplo
<code>&lt;</code>	menor	<code>inet '192.168.1.5' &lt; inet '192.168.1.6'</code>
<code>&lt;=</code>	menor ou igual	<code>inet '192.168.1.5' &lt;= inet '192.168.1.5'</code>
<code>=</code>	igual	<code>inet '192.168.1.5' = inet '192.168.1.5'</code>
<code>&gt;=</code>	maior ou igual	<code>inet '192.168.1.5' &gt;= inet '192.168.1.5'</code>
<code>&gt;</code>	maior	<code>inet '192.168.1.5' &gt; inet '192.168.1.4'</code>
<code>&lt;&gt;</code>	diferente	<code>inet '192.168.1.5' &lt;&gt; inet '192.168.1.4'</code>
<code>&lt;&lt;</code>	está contido em	<code>inet '192.168.1.5' &lt;&lt; inet '192.168.1/24'</code>
<code>&lt;=&lt;</code>	está contido ou é igual	<code>inet '192.168.1/24' &lt;=&lt; inet '192.168.1/24'</code>
<code>&gt;&gt;</code>	contém	<code>inet '192.168.1/24' &gt;&gt; inet '192.168.1.5'</code>
<code>&gt;=&gt;</code>	contém ou é igual	<code>inet '192.168.1/24' &gt;=&gt; inet '192.168.1/24'</code>

A Tabela 9-34 mostra as funções disponíveis para uso com os tipos de dado `cidr` e `inet`. As funções `host()`, `text()` e `abbrev()` se destinam, principalmente, a oferecer formatos de exibição alternativos. Um valor texto pode ser convertido em `inet` utilizando a sintaxe normal de conversão: `inet(expressão)` ou `nome_da_coluna::inet`.

Tabela 9-34. Funções para os tipos `cidr` e `inet`

Função	Tipo retornado	Descrição	Exemplo	Resultado
<code>broadcast(inet)</code>	<code>inet</code>	endereço de difusão da rede	<code>broadcast('192.168.1.5/24')</code>	<code>192.168.1.255/24</code>
<code>host(inet)</code>	<code>text</code>	extraí o endereço de IP como texto	<code>host('192.168.1.5/24')</code>	<code>192.168.1.5</code>
<code>masklen(inet)</code>	<code>integer</code>	extraí o comprimento da	<code>masklen(</code>	<code>24</code>

Função	Tipo retornado	Descrição	Exemplo	Resultado
		máscara de rede	'192.168.1.5/24')	
set_masklen(inet, integer)	inet	define o comprimento da máscara de rede para o valor do tipo inet	set_masklen('192.168.1.5/24', 16)	192.168.1.5/16
netmask(inet)	inet	constrói máscara de rede para a rede	netmask('192.168.1.5/24')	255.255.255.0
hostmask(inet)	inet	constrói máscara de hospedeiro para a rede	hostmask('192.168.23.20/30')	0.0.0.3
network(inet)	cidr	extrai do endereço a parte de rede	network('192.168.1.5/24')	192.168.1.0/24
text(inet)	text	extrai o endereço de IP e o comprimento da máscara de rede como texto	text(inet '192.168.1.5')	192.168.1.5/32
abbrev(inet)	text	formato de exibição abreviado como texto	abbrev(cidr '10.1.0.0/16')	10.1/16
family(inet)	integer	extrai a família de endereços; 4 para IPv4, e 6 para IPv6	family('::1')	6

A Tabela 9-35 mostra as funções disponíveis para uso com o tipo `macaddr`. A função `trunc(macaddr)` retorna o endereço MAC com os últimos 3 bytes tornados zero, podendo ser utilizado para associar o prefixo remanescente com o fabricante. Na distribuição do código fonte o diretório `contrib/mac` contém alguns utilitários para criar e manter esta tabela de associação.<sup>14</sup>

**Tabela 9-35. Funções para o tipo `macaddr`**

Função	Tipo retornado	Descrição	Exemplo	Resultado
trunc(macaddr)	macaddr	torna os 3 últimos bytes iguais a zero	trunc(macaddr '12:34:56:78:90:ab')	12:34:56:00:00:00

O tipo `macaddr` também suporta os operadores relacionais padrão (>, <=, etc.) para a ordenação lexicográfica.

## 9.12. Funções para manipulação de seqüências

Esta seção descreve as funções do PostgreSQL que operam com *objetos de seqüência*. Os objetos de seqüência (também chamados de geradores de seqüência, ou simplesmente de seqüências), são tabelas especiais de uma única linha criadas pelo comando `CREATE SEQUENCE`. O objeto de seqüência é usado normalmente para gerar identificadores únicos para linhas de tabelas. As funções de seqüência, listadas na Tabela 9-36, fornecem métodos simples, seguros para multiusuários, para obter valores sucessivos da seqüência a partir dos objetos de seqüência.

**Tabela 9-36. Funções de seqüência**

Função	Tipo retornado	Descrição
nextval(text)	bigint	Avança a seqüência e retorna o novo valor
currval(text)	bigint	Retorna o valor obtido mais recentemente por <code>nextval</code>

Função	Tipo retornado	Descrição
<code>setval(text, bigint)</code>	<code>bigint</code>	Define o valor corrente da sequência
<code>setval(text, bigint, boolean)</code>	<code>bigint</code>	Define o valor corrente da sequência e o sinalizador <code>is_called</code>

Por motivos históricos, a sequência a ser operada pela chamada da função de sequência é especificada por um argumento que é um texto em uma cadeia de caracteres. Para obter alguma compatibilidade com o tratamento usual dos nomes SQL, a função de sequência converte as letras do argumento em minúsculas, a não ser que a cadeia de caracteres esteja entre aspas. Portanto

```
nextval('foo')      opera na sequência foo
nextval('FOO')      opera na sequência foo
nextval('"Foo"')    opera na sequência Foo
```

Havendo necessidade, o nome da sequência pode ser qualificado pelo esquema:

```
nextval('meu_esquema.foo')  opera em meu_esquema.foo
nextval('"meu_esquema".foo') o mesmo acima
nextval('foo')              procura por foo no caminho de procura
```

É claro que o texto do argumento pode ser o resultado de uma expressão, e não somente um literal simples, o que algumas vezes é útil.

As funções de sequência disponíveis são:

`nextval`

Avança o objeto de sequência para seu próximo valor e retorna este valor. Isto é feito atomicamente: mesmo que várias sessões executem `nextval` simultaneamente, cada uma recebe um valor distinto da sequência com segurança.

`currval`

Retorna o valor retornado mais recentemente por `nextval` para esta sequência na sessão corrente (relata erro se `nextval` nunca tiver sido chamada para esta sequência nesta sessão). Deve ser observado que como é retornado o valor da sessão local, uma resposta previsível é obtida mesmo que outras sessões tenham executado `nextval` após a sessão corrente tê-lo feito.

`setval`

Redefine o valor do contador do objeto de sequência. A forma com dois parâmetros define o campo `last_value` (último valor) da sequência com o valor especificado, e define o campo `is_called` como `true`, indicando que o próximo `nextval` avançará a sequência antes de retornar o valor. Na forma com três parâmetros, `is_called` pode ser definido tanto como `true` quanto como `false`. Se for definido como `false`, o próximo `nextval` retornará o próprio valor especificado, e o avanço da sequência somente começará no `nextval` seguinte. Por exemplo,

```
SELECT setval('foo', 42);           o próximo nextval() retorna 43
SELECT setval('foo', 42, true);      o mesmo acima
SELECT setval('foo', 42, false);     o próximo nextval() retorna 42
```

O resultado retornado por `setval` é simplesmente o valor de seu segundo argumento.

**Importante:** Para evitar o bloqueio de transações simultâneas que obtêm números de uma mesma sequência, a operação `nextval` nunca é desfeita (`rolled back`); ou seja, após o valor ser obtido este passa a ser considerado como tendo sido usado, mesmo que a transação que executou o `nextval` seja interrompida posteriormente. Isto significa que as transações interrompidas podem deixar “buracos” não utilizados na sequência de valores atribuídos. Além disso, as operações `setval` nunca são desfeitas, também.

Se o objeto de sequência tiver sido criado com os parâmetros padrão, as chamadas à `nextval()` retornam valores sucessivos começando por um. Outros comportamentos podem ser obtidos utilizando parâmetros especiais no comando `CREATE SEQUENCE`; consulte a página de referência deste comando para obter informações adicionais.

## 9.13. Expressões condicionais

Esta seção descreve as expressões condicionais em conformidade com o SQL disponíveis no PostgreSQL.

**Dica:** Havendo alguma necessidade não atendida pelas funcionalidades destas expressões condicionais, deve ser considerado o desenvolvimento de um procedimento armazenado usando uma linguagem de programação com mais recursos.

### 9.13.1. CASE

A expressão `CASE` do SQL é uma expressão condicional genérica, semelhante às declarações `if/else` de outras linguagens:

```
CASE WHEN condição THEN resultado
      [WHEN ...]
      [ELSE resultado]
END
```

A cláusula `CASE` pode ser usada em qualquer lugar onde uma expressão for válida. A *condição* é uma expressão que retorna um resultado boolean. Se o resultado for verdade, então o valor da expressão `CASE` é o *resultado* que segue a condição. Se o resultado for falso, todas as cláusulas `WHEN` seguintes são analisadas da mesma maneira. Se o resultado de nenhuma *condição* `WHEN` for verdade, então o valor da expressão `CASE` é o valor do *resultado* na cláusula `ELSE`. Se a cláusula `ELSE` for omitida, e nenhuma condição for satisfeita, o resultado será nulo.

Exemplo:

```
=> SELECT * FROM teste;

a
---
1
2
3

=> SELECT a,
->     CASE WHEN a=1 THEN 'um'
->           WHEN a=2 THEN 'dois'
->           ELSE 'outro'
->     END AS caso
-> FROM teste;

a | caso
---+-----
1 | um
2 | dois
3 | outro
```

Os tipos de dado de todas as expressões *resultado* devem ser conversíveis em um único tipo de dado de saída. Consulte a Seção 10.5 para obter mais detalhes.

A expressão `CASE` “simplificada”, mostrada abaixo, é uma variante especializada da forma geral mostrada acima <sup>15 16</sup> :

```
CASE expressão
      WHEN valor THEN resultado
      [WHEN ...]
      [ELSE resultado]
END
```

A *expressão* é computada e comparada com todas as especificações de *valor* nas cláusulas `WHEN`, até encontrar um que seja igual. Se não for encontrado nenhum valor igual, é retornado o *resultado* na cláusula `ELSE` (ou o valor nulo). Esta forma é semelhante à declaração `switch` da linguagem C.

O exemplo mostrado acima pode ser escrito utilizando a sintaxe simplificada da expressão CASE:

```
=> SELECT a,
->     CASE a WHEN 1 THEN 'um'
->           WHEN 2 THEN 'dois'
->           ELSE 'outro'
->     END AS caso
-> FROM teste;
```

```
a | caso
---+-----
1 | um
2 | dois
3 | outro
```

A expressão CASE não processa nenhuma subexpressão que não seja necessária para determinar o resultado. Por exemplo, esta é uma forma possível de evitar o erro gerado pela divisão por zero:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

### 9.13.2. COALESCE

```
COALESCE(valor [, ...])
```

A função COALESCE retorna o primeiro de seus argumentos que não for nulo. Só retorna nulo quando todos os seus argumentos são nulos. Geralmente é útil para substituir o valor padrão quando este é o valor nulo, quando os dados são usados para exibição. Por exemplo:

```
SELECT coalesce(descrição, descrição_curta, '(nenhuma)') ...
```

Como a expressão CASE, a função COALESCE não processa os argumentos que não são necessários para determinar o resultado, ou seja, os argumentos à direita do primeiro argumento que não for nulo não são avaliados.

### 9.13.3. NULLIF

```
NULLIF(valor1, valor2)
```

A função NULLIF retorna o valor nulo se, e somente se, *valor1* e *valor2* forem iguais. Senão, retorna *valor1*. Pode ser utilizada para realizar a operação inversa do exemplo para COALESCE mostrado acima:

```
SELECT nullif(valor, '(nenhuma)') ...
```

#### Exemplo 9-15. Inserir nulo quando a cadeia de caracteres estiver vazia

Neste exemplo são utilizadas as funções NULLIF e TRIM para inserir o valor nulo na coluna da tabela quando a cadeia de caracteres passada como parâmetro para o comando INSERT preparado estiver vazia ou só contiver espaços.<sup>17</sup>

```
=> CREATE TEMPORARY TABLE t (c1 SERIAL PRIMARY KEY, c2 TEXT);
=> PREPARE inserir (TEXT) AS
->     INSERT INTO t VALUES(DEFAULTT, nullif(trim(' ' from $1),''));
=> EXECUTE inserir('linha 1');
=> EXECUTE inserir('');
=> EXECUTE inserir(' ');
=> EXECUTE inserir(NULL);
=> \pset null (nulo)
=> SELECT * FROM t;
```

```
c1 | c2
---+-----
1 | linha 1
2 | (nulo)
3 | (nulo)
4 | (nulo)
(4 linhas)
```

## 9.14. Funções e operadores para matrizes

A Tabela 9-37 mostra os operadores disponíveis para o tipo `array`.

**Tabela 9-37. Operadores para o tipo `array`**

Operador	Descrição	Exemplo	Resultado
<code>=</code>	igual	<code>ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]</code>	t
<code>&lt;&gt;</code>	diferente	<code>ARRAY[1,2,3] &lt;&gt; ARRAY[1,2,4]</code>	t
<code>&lt;</code>	menor	<code>ARRAY[1,2,3] &lt; ARRAY[1,2,4]</code>	t
<code>&gt;</code>	maior	<code>ARRAY[1,4,3] &gt; ARRAY[1,2,4]</code>	t
<code>&lt;=</code>	menor ou igual	<code>ARRAY[1,2,3] &lt;= ARRAY[1,2,3]</code>	t
<code>&gt;=</code>	maior ou igual	<code>ARRAY[1,4,3] &gt;= ARRAY[1,4,3]</code>	t
<code>  </code>	concatenação de matriz com matriz	<code>ARRAY[1,2,3]    ARRAY[4,5,6]</code>	<code>{1,2,3,4,5,6}</code>
<code>  </code>	concatenação de matriz com matriz	<code>ARRAY[1,2,3]    ARRAY[[4,5,6],[7,8,9]]</code>	<code>{{1,2,3},{4,5,6},{7,8,9}}</code>
<code>  </code>	concatenação de elemento com matriz	<code>3    ARRAY[4,5,6]</code>	<code>{3,4,5,6}</code>
<code>  </code>	concatenação de matriz com elemento	<code>ARRAY[4,5,6]    7</code>	<code>{4,5,6,7}</code>

Consulte a Seção 8.10 para obter informações adicionais sobre o comportamento dos operadores de matriz.

A Tabela 9-38 mostra as funções disponíveis para uso com o tipo `array`. Veja na Seção 8.10 uma discussão mais detalhada e exemplos de uso destas funções.

**Tabela 9-38. Funções para o tipo `array`**

Função	Tipo retornado	Descrição	Exemplo	Resultado
<code>array_cat</code> ( <code>anyarray</code> , <code>anyarray</code> )	<code>anyarray</code>	concatena duas matrizes	<code>array_cat(ARRAY[1,2,3], ARRAY[4,5])</code>	<code>{1,2,3,4,5}</code>
<code>array_append</code> ( <code>anyarray</code> , <code>anyelement</code> )	<code>anyarray</code>	anexa um elemento no final de uma matriz	<code>array_append(ARRAY[1,2], 3)</code>	<code>{1,2,3}</code>
<code>array_prepend</code> ( <code>anyelement</code> , <code>anyarray</code> )	<code>anyarray</code>	anexa um elemento no início de uma matriz	<code>array_prepend(1, ARRAY[2,3])</code>	<code>{1,2,3}</code>
<code>array_dims</code> ( <code>anyarray</code> )	<code>text</code>	retorna a representação textual das dimensões da matriz	<code>array_dims(array[[1,2,3], [4,5,6]])</code>	<code>[1:2][1:3]</code>
<code>array_lower</code> ( <code>anyarray</code> , <code>integer</code> )	<code>integer</code>	retorna o limite inferior da dimensão especificada da matriz	<code>array_lower(array_prepend(0, ARRAY[1,2,3]), 1)</code>	0



Função	Tipo retornado	Descrição	Exemplo	Resultado
<code>array_upper</code> ( <code>anyarray</code> , <code>integer</code> )	<code>integer</code>	retorna o limite superior da dimensão especificada da matriz	<code>array_upper(ARRAY[1,2,3,4], 1)</code>	4
<code>array_to_string</code> ( <code>anyarray</code> , <code>text</code> )	<code>text</code>	concatena os elementos da matriz utilizando o delimitador especificado	<code>array_to_string(array[1, 2, 3], '~^~')</code>	<code>1~^~2~^~3</code>
<code>string_to_array</code> ( <code>text</code> , <code>text</code> )	<code>text[]</code>	divide uma cadeia de caracteres em elementos de matriz utilizando o delimitador especificado	<code>string_to_array('xx~^~yy~^~zz', '~^~')</code>	{xx,yy,zz}

## 9.15. Funções de agregação

As *funções de agregação* retornam um único valor como resultado de um conjunto de valores de entrada. A Tabela 9-39 mostra as funções de agregação internas. As considerações especiais sobre a sintaxe das funções de agregação são explicadas na Seção 4.2.7. Consulte a Seção 2.7 para obter informações introdutórias adicionais.

**Tabela 9-39. Funções de agregação**

Função	Tipo do argumento	Tipo retornado	Descrição
<code>avg(expressão)</code>	<code>smallint</code> , <code>integer</code> , <code>bigint</code> , <code>real</code> , <code>double precision</code> , <code>numeric</code> ou <code>interval</code>	<code>numeric</code> para qualquer argumento de tipo inteiro, <code>double precision</code> para argumento de tipo ponto flutuante, caso contrário o mesmo tipo de dado do argumento	a média (média aritmética) de todos os valores de entrada
<code>bit_and(expressão)</code>	<code>smallint</code> , <code>integer</code> , <code>bigint</code> ou <code>bit</code>	o mesmo tipo de dado do argumento	o AND bit a bit de todos os valores de entrada não nulos, ou nulo se algum for nulo
<code>bit_or(expressão)</code>	<code>smallint</code> , <code>integer</code> , <code>bigint</code> ou <code>bit</code>	o mesmo tipo de dado do argumento	o OR bit a bit de todos os valores de entrada não nulos, ou nulo se algum for nulo
<code>bool_and(expressão)</code>	<code>bool</code>	<code>bool</code>	verdade se todos os valores de entrada forem verdade, senão falso
<code>bool_or(expressão)</code>	<code>bool</code>	<code>bool</code>	verdade se ao menos um dos valores de entrada for verdade, senão falso
<code>count(*)</code>		<code>bigint</code>	número de valores de entrada
<code>count(expressão)</code>	<code>any</code>	<code>bigint</code>	número de valores de

Função	Tipo do argumento	Tipo retornado	Descrição
			entrada para os quais o valor da <i>expressão</i> não é nulo
<code>every(expressão)</code>	bool	bool	equivale ao <code>bool_and</code>
<code>max(expressão)</code>	qualquer tipo de dado matriz, numérico, cadeia de caracteres, data ou hora	o mesmo tipo de dado do argumento	valor máximo da <i>expressão</i> entre todos os valores de entrada
<code>min(expressão)</code>	qualquer tipo de dado matriz, numérico, cadeia de caracteres, data ou hora	o mesmo tipo de dado do argumento	valor mínimo da <i>expressão</i> entre todos os valores de entrada
<code>stddev(expressão)</code>	smallint, integer, bigint, real, double precision ou numeric	double precision para argumentos de ponto flutuante, caso contrário numeric.	desvio padrão da amostra dos valores de entrada
<code>sum(expressão)</code>	smallint, integer, bigint, real, double precision, numeric ou interval	bigint para argumentos smallint ou integer, numeric para argumentos bigint, double precision para argumentos de ponto flutuante, caso contrário o mesmo tipo de dado do argumento	somatório da <i>expressão</i> para todos os valores de entrada
<code>variance(expressão)</code>	smallint, integer, bigint, real, double precision ou numeric	double precision para argumentos de ponto flutuante, caso contrário numeric.	variância da amostra dos valores de entrada (quadrado do desvio padrão da amostra)

Deve ser observado que, com exceção do `count`, estas funções retornam o valor nulo quando nenhuma linha for selecionada. Em particular, `sum` de nenhuma linha retorna nulo, e não zero como poderia ser esperado. A função `coalesce` pode ser utilizada para substituir nulo por zero quando for necessário.

**Nota:** As agregações booleanas `bool_and` e `bool_or` correspondem às agregações do padrão SQL `every` e `any` ou `some`. Para `any` e `some` parece haver uma ambigüidade na sintaxe do padrão:

```
SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
```

Neste caso `ANY` pode ser considerada tanto como levando a uma subconsulta quanto a uma agregação, se a expressão de seleção retornar 1 linha. Portanto, não pode ser dado o nome padrão a estas agregações.

**Nota:** Os usuários acostumados a trabalhar com outros sistemas gerenciadores de banco de dados SQL podem ficar surpresos com as características de desempenho de certas funções de agregação do PostgreSQL, quando a agregação é aplicada a toda a tabela (em outras palavras, não é especificada nenhuma cláusula `WHERE`). Em particular, uma consulta como

```
SELECT min(col) FROM alguma_tabela;
```

será executada pelo PostgreSQL usando a varredura seqüencial de toda a tabela. Outros sistemas de banco de dados podem otimizar as consultas deste tipo utilizando um índice na coluna, caso esteja disponível. De maneira semelhante, as funções de agregação `max()` e `count()` sempre requerem que seja aplicada uma varredura seqüencial em toda a tabela no PostgreSQL.

O PostgreSQL não pode implementar facilmente esta otimização, porque também permite consultas em agregações definidas pelo usuário. Uma vez que `min()`, `max()` e `count()` são definidas usando uma API genérica para funções de agregação, não há dispositivo para executar casos especiais destas funções sob certas circunstâncias.

Felizmente existe um recurso simples para contornar os problemas com `min()` e `max()`. A consulta mostrada abaixo é equivalente à consulta acima, exceto que tira vantagem de um índice `B-tree`, caso algum esteja presente na coluna em questão.

```
SELECT col FROM alguma_tabela ORDER BY col ASC LIMIT 1;
```

Pode ser utilizada Uma consulta semelhante (obtida trocando `DESC` por `ASC` na consulta acima) no lugar de `max()`.

Infelizmente, não existe uma consulta trivial semelhante que possa ser utilizada para melhorar o desempenho do `count()` aplicado a toda a tabela.

## 9.16. Expressões de subconsulta

Esta seção descreve as expressões de subconsulta em conformidade com o padrão SQL disponíveis no PostgreSQL. Todas as formas das expressões documentadas nesta seção retornam resultados booleanos (verdade/falso).

### 9.16.1. EXISTS

```
EXISTS ( subconsulta )
```

O argumento do `EXISTS` é uma declaração `SELECT` arbitrária, ou uma *subconsulta*. A subconsulta é processada para determinar se retorna alguma linha. Se retornar pelo menos uma linha, o resultado de `EXISTS` é “verdade”; se a subconsulta não retornar nenhuma linha, o resultado de `EXISTS` é “falso”.

A subconsulta pode referenciar variáveis da consulta que a envolve, que atuam como constantes durante a execução da subconsulta.

A subconsulta geralmente só é processada até ser determinado se retorna pelo menos uma linha, e não até o fim. Não é razoável escrever uma subconsulta que tenha efeitos colaterais (tal como chamar uma função de sequência); pode ser difícil prever se o efeito colateral ocorrerá ou não.

Como o resultado depende apenas de alguma linha ser retornada, e não do conteúdo da linha, normalmente não há interesse na saída da subconsulta. Uma convenção de codificação habitual é escrever todos os testes de `EXISTS` na forma `EXISTS(SELECT 1 WHERE ...)`. Entretanto, existem exceções para esta regra, como as subconsultas que utilizam `INTERSECT`.

Este exemplo simples é como uma junção interna em `col2`, mas produz no máximo uma linha de saída para cada linha de `tab1`, mesmo havendo várias linhas correspondentes em `tab2`:

```
SELECT col1 FROM tab1
WHERE EXISTS(SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

#### Exemplo 9-16. Utilização das cláusulas CASE e EXISTS juntas

Neste exemplo a tabela `frutas` é consultada para verificar se o alimento é uma fruta ou não. Caso o alimento conste da tabela `frutas` é uma fruta, caso não conste não é uma fruta. Abaixo está mostrado o script utilizado para criar e carregar as tabelas e executar a consulta.<sup>18</sup>

```
CREATE TEMPORARY TABLE frutas (id SERIAL PRIMARY KEY, nome TEXT);
INSERT INTO frutas VALUES (DEFAULT, 'banana');
INSERT INTO frutas VALUES (DEFAULT, 'maçã');
CREATE TEMPORARY TABLE alimentos (id SERIAL PRIMARY KEY, nome TEXT);
INSERT INTO alimentos VALUES (DEFAULT, 'maçã');
INSERT INTO alimentos VALUES (DEFAULT, 'espinafre');
SELECT nome, CASE WHEN EXISTS (SELECT nome FROM frutas WHERE nome=a.nome)
                  THEN 'sim'
                  ELSE 'não'
                END AS fruta
FROM alimentos a;
```

Abaixo está mostrado o resultado da execução do script.

```

nome      | fruta
-----+-----
maçã      | sim
espinafre | não
(2 linhas)

```

### 9.16.2. IN

*expressão* IN (*subconsulta*)

O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é processada e comparada com cada linha do resultado da subconsulta. O resultado do IN é “verdade” se for encontrada uma linha igual na subconsulta. O resultado é “falso” se não for encontrada nenhuma linha igual (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

Deve ser observado que, se o resultado da expressão à esquerda for nulo, ou se não houver nenhum valor igual à direita e uma das linhas à direita tiver o valor nulo, o resultado da construção IN será nulo, e não falso. Isto está de acordo com as regras normais do SQL para combinações booleanas de valores nulos.

Da mesma forma que no EXISTS, não é razoável assumir que a subconsulta será processada até o fim.

*construtor\_de\_linha* IN (*subconsulta*)

O lado esquerdo desta forma do IN é um construtor de linha, conforme descrito na Seção 4.2.11. O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente tantas colunas quantas forem as expressões na linha do lado esquerdo. As expressões do lado esquerdo são processadas e comparadas, por toda a largura, com cada linha do resultado da subconsulta. O resultado do IN é “verdade” se for encontrada uma linha igual na subconsulta. O resultado é “falso” se não for encontrada nenhuma linha igual (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

Da maneira usual, os valores nulos nas linhas são combinados de acordo com as regras normais para expressões booleana do SQL. As linhas são consideradas iguais se todos os seus membros correspondentes forem não-nulos e iguais; as linhas são diferentes se algum membro correspondente for não-nulo e diferente; senão o resultado da comparação é desconhecido (nulo). Se o resultado de todas as linhas for diferente ou nulo, com pelo menos um nulo, o resultado do IN será nulo.

#### Exemplo 9-17. Utilização das cláusulas CASE e IN juntas

Este exemplo é idêntico ao Exemplo 9-16, só que utiliza a cláusula IN para executar a consulta, conforme mostrado abaixo.<sup>19</sup>

```

SELECT nome, CASE WHEN nome IN (SELECT nome FROM frutas)
                  THEN 'sim'
                  ELSE 'não'
                END AS fruta
FROM alimentos;

```

Abaixo está mostrado o resultado da execução do script.

```

nome      | fruta
-----+-----
maçã      | sim
espinafre | não
(2 linhas)

```

### 9.16.3. NOT IN

*expressão* NOT IN (*subconsulta*)

O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é processada e comparada com cada linha do resultado da subconsulta. O resultado de NOT IN é “verdade” se somente forem encontradas linhas diferentes na subconsulta (incluindo o caso especial onde a subconsulta não retorna nenhuma linha). O resultado é “falso” se for encontrada alguma linha igual.

Deve ser observado que se o resultado da expressão à esquerda for nulo, ou se não houver nenhum valor igual à direita e uma das linhas à direita tiver o valor nulo, o resultado da construção `NOT IN` será nulo, e não verdade. Isto está de acordo com as regras normais do SQL para combinações booleanas de valores nulos.

Da mesma forma que no `EXISTS`, não é razoável assumir que a subconsulta será processada até o fim.

```
construtor_de_linha NOT IN (subconsulta)
```

O lado esquerdo desta forma do `NOT IN` é um construtor de linha, conforme descrito na Seção 4.2.11. O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente tantas colunas quantas forem as expressões na linha do lado esquerdo. As expressões do lado esquerdo são processadas e comparadas, por toda a largura, com cada linha do resultado da subconsulta. O resultado do `NOT IN` é “verdade” se somente forem encontradas linhas diferentes na subconsulta (incluindo o caso especial onde a subconsulta não retorna nenhuma linha). O resultado é “falso” se for encontrada alguma linha igual.

Da maneira usual, os valores nulos nas linhas são combinados de acordo com as regras normais para expressões booleana do SQL. As linhas são consideradas iguais se todos os seus membros correspondentes forem não-nulos e iguais; as linhas são diferentes se algum membro correspondente for não-nulo e diferente; senão o resultado da comparação é desconhecido (nulo). Se o resultado de todas as linhas for diferente ou nulo, com pelo menos um nulo, o resultado do `NOT IN` será nulo.

#### 9.16.4. ANY/SOME

```
expressão operador ANY (subconsulta)
```

```
expressão operador SOME (subconsulta)
```

O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é processada e comparada com cada linha do resultado da subconsulta usando o *operador* especificado, devendo produzir um resultado booleano. O resultado do `ANY` é “verdade” se for obtido algum resultado verdade. O resultado é “falso” se nenhum resultado verdade for encontrado (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).<sup>20</sup>

`SOME` é sinônimo de `ANY`. `IN` equivale a `= ANY`.

Deve ser observado que se não houver nenhuma comparação bem sucedida, e pelo menos uma linha da direita gerar nulo como resultado do operador, o resultado da construção `ANY` será nulo, e não falso. Isto está de acordo com as regras normais do SQL para combinações booleanas de valores nulos.

Do mesmo modo que no `EXISTS`, não é razoável supor que a subconsulta será processada até o fim.

```
construtor_de_linha operador ANY (subconsulta)
```

```
construtor_de_linha operador SOME (subconsulta)
```

O lado esquerdo desta forma do `ANY` é um construtor de linha, conforme descrito na Seção 4.2.11. O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente tantas colunas quantas forem as expressões existentes na linha do lado esquerdo. As expressões do lado esquerdo são processadas e comparadas, por toda a largura, com cada linha do resultado da subconsulta utilizando o *operador* especificado. Atualmente, somente são permitidos os operadores `=` e `<>` em construções `ANY` para toda a largura da linha. O resultado do `ANY` é “verdade” se for encontrada alguma linha igual ou diferente, respectivamente. O resultado será “falso” se não for encontrada nenhuma linha deste tipo (incluindo o caso especial onde a subconsulta não retorna nenhuma linha).

Da maneira usual, os valores nulos nas linhas são combinados de acordo com as regras normais para expressões booleana do SQL. As linhas são consideradas iguais se todos os seus membros correspondentes forem não-nulos e iguais; as linhas são diferentes se algum membro correspondente for não-nulo e diferente; senão o resultado da comparação é desconhecido (nulo). Se houver pelo menos um resultado de linha nulo, então o resultado de `ANY` não poderá ser falso; será verdade ou nulo.

#### Exemplo 9-18. Utilização das cláusulas `CASE` e `ANY` juntas

Este exemplo é idêntico ao Exemplo 9-16, só que utiliza a cláusula `ANY` para executar a consulta, conforme mostrado abaixo.<sup>21</sup>

```
SELECT nome, CASE WHEN nome = ANY (SELECT nome FROM frutas)
                THEN 'sim'
                ELSE 'não'
            END AS fruta
FROM alimentos;
```

Abaixo está mostrado o resultado da execução do script.

```
nome      | fruta
-----+-----
maçã      | sim
espinafre | não
(2 linhas)
```

### 9.16.5. ALL

*expressão operador ALL (subconsulta)*

O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente uma coluna. A expressão à esquerda é processada e comparada com cada linha do resultado da subconsulta usando o *operador* especificado, devendo produzir um resultado booleano. O resultado do ALL é “verdade” se o resultado de todas as linhas for verdade (incluindo o caso especial onde a subconsulta não retorna nenhuma linha). O resultado é “falso” se for encontrado algum resultado falso.

NOT IN equivale a <> ALL.

Deve ser observado que se todas as comparações forem bem-sucedidas, mas pelo menos uma linha da direita gerar nulo como resultado do operador, o resultado da construção ALL será nulo, e não verdade. Isto está de acordo com as regras normais do SQL para combinações booleanas de valores nulos.

Do mesmo modo que no EXISTS, não é razoável supor que a subconsulta será processada até o fim.

*construtor\_de\_linha operador ALL (subconsulta)*

O lado esquerdo desta forma do ALL é um construtor de linha, conforme descrito na Seção 4.2.11. O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente tantas colunas quantas forem as expressões existentes na linha do lado esquerdo. As expressões do lado esquerdo são processadas e comparadas, por toda a largura, com cada linha do resultado da subconsulta utilizando o *operador* especificado. Atualmente, somente são permitidos os operadores = e <> em construções ALL para toda a largura da linha. O resultado do ALL é “verdade” se todas as linhas da subconsulta forem iguais ou diferentes, respectivamente (incluindo o caso especial onde a subconsulta não retorna nenhuma linha). O resultado será “falso” se não for encontrada nenhuma linha que seja igual ou diferente, respectivamente.

Da maneira usual, os valores nulos nas linhas são combinados de acordo com as regras normais para expressões booleana do SQL. As linhas são consideradas iguais se todos os seus membros correspondentes forem não-nulos e iguais; as linhas são diferentes se algum membro correspondente for não-nulo e diferente; senão o resultado da comparação é desconhecido (nulo). Se houver pelo menos um resultado de linha nulo, então o resultado de ALL não poderá ser verdade; será falso ou nulo.

### 9.16.6. Comparação de toda a linha

*construtor\_de\_linha operador (subconsulta)*

O lado esquerdo é um construtor de linha, conforme descrito na Seção 4.2.11. O lado direito é uma subconsulta entre parênteses, que deve retornar exatamente tantas colunas quantas forem as expressões existentes na linha do lado esquerdo. Além disso, a subconsulta não pode retornar mais de uma linha (se não retornar nenhuma linha o resultado será considerado nulo). As expressões do lado esquerdo são processadas e comparadas, por toda a largura, com cada linha do resultado da subconsulta. Atualmente, somente são permitidos os operadores = e <> para comparação por toda a largura da linha. O resultado é “verdade” se as duas linhas forem iguais ou diferentes, respectivamente

Da maneira usual, os valores nulos nas linhas são combinados de acordo com as regras normais para expressões booleana do SQL. As linhas são consideradas iguais se todos os seus membros correspondentes forem não-nulos e iguais; as linhas são diferentes se algum membro correspondente for não-nulo e diferente; senão o resultado da comparação é desconhecido (nulo).

## 9.17. Comparações de linha e de matriz

Esta seção descreve várias construções especializadas para fazer comparações múltiplas entre grupos de valores. Estas formas são relacionadas sintaticamente com as formas das subconsultas da seção anterior, mas não envolvem subconsultas. As formas envolvendo subexpressões de matrizes são extensões do PostgreSQL; o restante está em conformidade com o padrão SQL. Todas as formas de expressão documentadas nesta seção retornam resultados booleanos (verdade/falso).

### 9.17.1. IN

*expressão* IN (*valor*[, ...])

O lado direito é uma lista de expressões escalares entre parênteses. O resultado é “verdade” se o resultado da expressão à esquerda for igual a qualquer uma das expressões à direita. Esta é uma notação abreviada de

```
expressão = valor1
OR
expressão = valor2
OR
...
```

Deve ser observado que se o resultado da expressão do lado esquerdo for nulo, ou se não houver valor igual do lado direito e pelo menos uma expressão do lado direito tiver resultado nulo, o resultado da construção IN será nulo, e não falso. Isto está de acordo com as regras normais do SQL para combinações booleanas de valores nulos.

### 9.17.2. NOT IN

*expressão* NOT IN (*valor*[, ...])

O lado direito é uma lista de expressões escalares entre parênteses. O resultado é “verdade” se o resultado da expressão à esquerda for diferente do resultado de todas as expressões à direita. Esta é uma notação abreviada de

```
expressão <> valor1
AND
expressão <> valor2
AND
...
```

Deve ser observado que se o resultado da expressão do lado esquerdo for nulo, ou se não houver valor igual do lado direito e pelo menos uma expressão do lado direito tiver resultado nulo, o resultado da construção NOT IN será nulo, e não verdade. Isto está de acordo com as regras normais do SQL para combinações booleanas de valores nulos.

**Dica:**  $x \text{ NOT IN } y$  equivale a  $\text{NOT } (x \text{ IN } y)$  para todos os casos. Entretanto, os valores nulos tem muito mais facilidade de confundir uma pessoa inexperiente trabalhando com NOT IN do que trabalhando com IN. É melhor expressar a condição na forma positiva se for possível.

### 9.17.3. ANY/SOME (matriz)

*expressão operador* ANY (*expressão\_de\_matriz*)  
*expressão operador* SOME (*expressão\_de\_matriz*)

O lado direito é uma expressão entre parênteses, que deve produzir um valor matriz. A expressão do lado esquerdo é processada e comparada com cada elemento da matriz utilizando o *operador* especificado, que deve produzir um resultado booleano. O resultado de ANY é “verdade” se for obtido algum resultado verdade. O resultado é “falso” se não for obtido nenhum resultado verdade (incluindo o caso especial onde a matriz possui zero elementos).

SOME é sinônimo de ANY.

### 9.17.4. ALL (matriz)

*expressão operador* ALL (*expressão\_de\_matriz*)

O lado direito é uma expressão entre parênteses, que deve produzir um valor matriz. A expressão do lado esquerdo é processada e comparada com cada elemento da matriz utilizando o *operador* especificado, que deve produzir um resultado booleano. O resultado de `ALL` é “verdade” se o resultado de todas as comparações for verdade (incluindo o caso especial onde a matriz possui zero elementos). O resultado é “falso” se for encontrado algum resultado falso.

### 9.17.5. Comparação por toda a largura da linha

*construtor\_de\_linha operador construtor\_de\_linha*

Cada um dos lados é um construtor de linha, conforme descrito na Seção 4.2.11. Os valores das duas linhas devem possuir o mesmo número de campos. Cada lado é avaliado e depois comparados por toda a largura da linha. Atualmente, somente são permitidos os operadores `=` e `<>` para comparação por toda a largura da linha. O resultado é “verdade” se as duas linhas forem iguais ou diferentes, respectivamente

Da maneira usual, os valores nulos nas linhas são combinados de acordo com as regras normais para expressões booleana do SQL. As linhas são consideradas iguais se todos os seus membros correspondentes forem não-nulos e iguais; as linhas são diferentes se algum membro correspondente for não-nulo e diferente; senão o resultado da comparação é desconhecido (nulo).

*construtor\_de\_linha IS DISTINCT FROM construtor\_de\_linha*

Esta construção é semelhante à comparação de linha `<>`, mas não retorna nulo para entradas nulas. Em vez disso, todos os valores nulos são considerados diferentes (distinto de) todos os valores não-nulos, e dois valores nulos são considerados iguais (não distintos). Portanto, o resultado será sempre verdade ou falso, e nunca nulo.

*construtor\_de\_linha IS NULL*  
*construtor\_de\_linha IS NOT NULL*

Estas construções testam um valor de linha para nulo e não nulo. Um valor de linha é considerado não-nulo se tiver pelo menos um campo que não seja nulo.

## 9.18. Funções que retornam conjunto

Esta seção descreve as funções que têm possibilidade de retornar mais de uma linha. Atualmente as únicas funções nesta classe são as funções geradoras de séries, conforme detalhado na Tabela 9-40.

**Tabela 9-40. Funções geradoras de séries**

Função	Tipo do argumento	Tipo retornado	Descrição
<code>generate_series( início, fim)</code>	int ou bigint	setof int ou setof bigint (o mesmo tipo do argumento)	Gera uma série de valores, do início ao fim, com tamanho do passo igual a um.
<code>generate_series( início, fim, passo)</code>	int ou bigint	setof int ou setof bigint (o mesmo tipo do argumento)	Gera uma série de valores, do início ao fim, com tamanho do passo igual a passo.

Quando o `passo` é positivo, são retornadas zero linhas se o `início` for maior que o `fim`. Inversamente, quando o `passo` é negativo são retornadas zero linhas se o `início` for menor que o `fim`. Também são retornadas zero linhas para entradas `NULL`. É errado o `passo` ser igual a zero. Alguns exemplos:

```
=> select * from generate_series(2,4);
```

```
generate_series
-----
                2
                3
                4
(3 linhas)
```



```
=> select * from generate_series(5,1,-2);
```

```
generate_series
-----
              5
              3
              1
(3 linhas)
```

```
=> select * from generate_series(4,3);
```

```
generate_series
-----
(0 linhas)
```

```
=> select current_date + s.a as dates from generate_series(0,14,7) as s(a);
```

```
dates
-----
2004-02-05
2004-02-12
2004-02-19
(3 linhas)
```

## 9.19. Funções de informação do sistema

A Tabela 9-41 mostra várias funções para extrair informações da sessão e do sistema.

**Tabela 9-41. Funções de informação da sessão**

Nome	Tipo retornado	Descrição
<code>current_database()</code>	name	nome do banco de dados corrente
<code>current_schema()</code>	name	nome do esquema corrente
<code>current_schemas(boolean)</code>	name[ ]	nomes dos esquemas no caminho de procura incluindo, opcionalmente, os esquemas implícitos
<code>current_user</code>	name	nome do usuário do contexto de execução corrente
<code>inet_client_addr()</code>	inet	endereço da conexão remota
<code>inet_client_port()</code>	int4	porta da conexão remota
<code>inet_server_addr()</code>	inet	endereço da conexão local
<code>inet_server_port()</code>	int4	port da conexão local
<code>session_user</code>	name	nome do usuário da sessão
<code>user</code>	name	equivale a <code>current_user</code>
<code>version()</code>	text	informação da versão do PostgreSQL

A função `session_user` normalmente retorna o usuário que iniciou a conexão com o banco de dados corrente; mas os superusuários podem mudar esta definição através de `SET SESSION AUTHORIZATION`. A função `current_user` retorna o identificador do usuário utilizado na verificação de permissão. Normalmente é igual ao usuário da sessão, mas muda durante a execução das funções com o atributo `SECURITY DEFINER`. Na terminologia Unix o usuário da sessão é o “usuário real”, enquanto o usuário corrente é o “usuário efetivo”.

**Nota:** As funções `current_user`, `session_user` e `user` possuem status sintático especial no SQL: devem ser chamadas sem parênteses no final.

A função `current_schema` retorna o nome do esquema que está à frente do caminho de procura (ou o valor nulo, se o caminho de procura estiver vazio). Este é o esquema utilizado na criação de qualquer tabela, ou outro objeto nomeado, sem especificar o esquema a ser utilizado. A função `current_schemas(boolean)` retorna uma matriz contendo os nomes de todos os esquemas presentes no caminho de procura. A opção booleana determina se os esquemas do sistema incluídos implicitamente, tal como `pg_catalog`, serão incluídos no caminho de procura retornado.

**Nota:** O caminho de procura pode ser alterado em tempo de execução. O comando é:

```
SET search_path TO esquema [, esquema, ...]
```

A função `inet_client_addr` retorna o endereço de IP do cliente corrente, e a função `inet_client_port` retorna o número da porta. A função `inet_server_addr` retorna o endereço de IP do servidor de destino da conexão corrente, e a função `inet_server_port` retorna o número da porta. Todas estas funções retornam NULL se a conexão corrente for através de um soquete do domínio Unix.

A função `version()` retorna uma cadeia de caracteres descrevendo a versão do servidor PostgreSQL.

A Tabela 9-42 mostra as funções que permitem o usuário consultar os privilégios de acesso dos objetos através de programa. Consulte a Seção 5.7 para obter informações adicionais sobre privilégios.

**Tabela 9-42. Funções de consulta a privilégios de acesso**

Nome	Tipo retornado	Descrição
<code>has_table_privilege (usuário, tabela, privilégio)</code>	boolean	o usuário possui o privilégio para a tabela
<code>has_table_privilege (tabela, privilégio)</code>	boolean	o usuário corrente possui o privilégio para a tabela
<code>has_database_privilege (usuário, banco_de_dados, privilégio)</code>	boolean	o usuário possui o privilégio para o banco de dados
<code>has_database_privilege (banco_de_dados, privilégio)</code>	boolean	o usuário corrente possui o privilégio para o banco de dados
<code>has_function_privilege (usuário, função, privilégio)</code>	boolean	o usuário possui o privilégio para a função
<code>has_function_privilege (função, privilégio)</code>	boolean	o usuário corrente possui o privilégio para a função
<code>has_language_privilege (usuário, linguagem, privilégio)</code>	boolean	o usuário possui o privilégio para a linguagem
<code>has_language_privilege (linguagem, privilégio)</code>	boolean	o usuário corrente possui o privilégio para a linguagem
<code>has_schema_privilege (usuário, esquema, privilégio)</code>	boolean	o usuário possui o privilégio para o esquema
<code>has_schema_privilege (esquema, privilégio)</code>	boolean	o usuário corrente possui o privilégio para o esquema
<code>has_tablespace_privilege (usuário, espaço_de_tabelas, privilégio)</code>	boolean	o usuário possui o privilégio para o espaço de tabelas
<code>has_tablespace_privilege (espaço_de_tabelas, privilégio)</code>	boolean	o usuário corrente possui o privilégio para o espaço de tabelas

A função `has_table_privilege` verifica se o usuário pode acessar a tabela de uma determinada maneira. O usuário pode ser especificado pelo nome ou pelo ID (`pg_user.usesysid`) ou, se o argumento for omitido, é assumido `current_user`. A tabela pode ser especificada pelo nome ou pelo OID. Portanto, na verdade existem seis variações de `has_table_privilege`, que podem ser distinguidas pelo número e tipo de seus argumentos. Quando se especifica pelo nome, o nome pode ser qualificado pelo esquema se for necessário. O tipo de privilégio de acesso desejado é especificado através de texto em uma cadeia de caracteres, que deve resultar em um dos seguintes valores: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `RULE`, `REFERENCES` ou `TRIGGER` (não faz diferença maiúscula ou minúsculas). Exemplo:

```
SELECT has_table_privilege('meu_esquema.minha_tabela', 'select');
```

A função `has_database_privilege` verifica se o usuário pode acessar o banco de dados de uma determinada maneira. As possibilidades para os argumentos são análogas às da função `has_table_privilege`. O tipo de privilégio de acesso desejado deve resultar em `CREATE`, `TEMPORARY` ou `TEMP` (que equivale a `TEMPORARY`).

A função `has_function_privilege` verifica se o usuário pode acessar a função de uma determinada maneira. As possibilidades para os argumentos são análogas às da função `has_table_privilege`. Ao se especificar a função através de um texto em uma cadeia de caracteres em vez de pelo OID, a entrada permitida é a mesma que para o tipo de dado `regprocedure` (consulte a Seção 8.12). O tipo de privilégio de acesso desejado deve resultar em `EXECUTE`. Exemplo:

```
SELECT has_function_privilege('joel', 'minha_função(int, text)', 'execute');
```

A função `has_language_privilege` verifica se o usuário pode acessar a linguagem procedural de uma determinada maneira. As possibilidades para os seus argumentos são análogas às da função `has_table_privilege`. O tipo de privilégio de acesso desejado deve resultar em `USAGE`.

A função `has_schema_privilege` verifica se o usuário pode acessar o esquema de uma determinada maneira. As possibilidades para os seus argumentos são análogas às da função `has_table_privilege`. O tipo de privilégio de acesso desejado deve resultar em `CREATE` or `USAGE`.

A função `has_tablespace_privilege` verifica se o usuário pode acessar o espaço de tabelas de uma determinada maneira. As possibilidades para os seus argumentos são análogas às da função `has_table_privilege`. O tipo de privilégio de acesso desejado deve resultar em `CREATE`.

Para verificar se o usuário possui a opção de concessão para um determinado privilégio, deve ser anexado `WITH GRANT OPTION` à palavra chave do privilégio; por exemplo `'UPDATE WITH GRANT OPTION'`.

A Tabela 9-43 mostra as funções que determinam se um certo objeto é *visível* através do caminho de procura de esquema corrente. Uma tabela é dita visível se o esquema que a contém está no caminho de procura e nenhuma tabela com o mesmo nome aparece antes no caminho de procura. Equivale a declarar que a tabela pode ser referenciada pelo seu nome sem uma qualificação de esquema explícita. Por exemplo, para listar o nome de todas as tabelas visíveis:

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

**Tabela 9-43. Funções de consulta à visibilidade do esquema**

Nome	Tipo retornado	Descrição
<code>pg_table_is_visible (oid_da_tabela)</code>	boolean	a tabela é visível no caminho de procura
<code>pg_type_is_visible (oid_do_tipo)</code>	boolean	o tipo (ou domínio) é visível no caminho de procura
<code>pg_function_is_visible (oid_da_função)</code>	boolean	a função é visível no caminho de procura
<code>pg_operator_is_visible (oid_do_operador)</code>	boolean	o operador é visível no caminho de procura
<code>pg_opclass_is_visible (oid_da_classe_de_operadores)</code>	boolean	a classe de operadores é visível no caminho de procura
<code>pg_conversion_is_visible (oid_da_conversão)</code>	boolean	a conversão é visível no caminho de procura

A função `pg_table_is_visible` realiza a verificação para as tabelas (ou visões, ou qualquer outro tipo de entrada em `pg_class`). As funções `pg_type_is_visible`, `pg_function_is_visible`, `pg_operator_is_visible`,

`pg_opclass_is_visible` e `pg_conversion_is_visible` realizam o mesmo tipo de verificação de visibilidade para os tipos (e domínios), funções, operadores, classes de operadores e conversões, respectivamente. Para as funções e os operadores, um objeto no caminho de procura é visível se não existir nenhum objeto com o mesmo nome *e mesmos tipos de dado dos argumentos* aparecendo antes no caminho. Para as classes de operadores, são considerados tanto o nome quanto o método de acesso de índice associado.

Todas estas funções requerem OIDs de objeto para identificar o objeto a ser verificado. Se for desejado verificar um objeto pelo seu nome, é conveniente utilizar os tipos aliás de OID (`regclass`, `regtype`, `regprocedure` ou `regoperator`). Exemplo:

```
SELECT pg_type_is_visible('meu_esquema.widget'::regtype);
```

Deve ser observado que não faz muito sentido testar um nome não qualificado desta maneira — se o nome puder ser reconhecido, então tem que ser visível.

A Tabela 9-44 mostra as funções que extraem informações dos catálogos do sistema.

**Tabela 9-44. Funções de informações dos catálogos do sistema**

Nome	Tipo retornado	Descrição
<code>format_type(oid_do_tipo, typemod)</code>	text	obtém o nome SQL do tipo de dado
<code>pg_get_viewdef(nome_da_visão)</code>	text	obtém o comando CREATE VIEW para a visão ( <i>em obsolescência</i> ) <sup>a</sup>
<code>pg_get_viewdef(nome_da_visão, impressão_alinhada)</code>	text	obtém o comando CREATE VIEW para a visão ( <i>em obsolescência</i> )
<code>pg_get_viewdef(oid_da_visão)</code>	text	obtém o comando CREATE VIEW para a visão
<code>pg_get_viewdef(oid_da_visão, impressão_alinhada)</code>	text	obtém o comando CREATE VIEW para a visão
<code>pg_get_ruledef(oid_da_regra)</code>	text	obtém o comando CREATE RULE para a regra
<code>pg_get_ruledef(oid_da_regra, impressão_alinhada)</code>	text	obtém o comando CREATE RULE para a regra
<code>pg_get_indexdef(oid_do_índice)</code>	text	obtém o comando CREATE INDEX para o índice
<code>pg_get_indexdef(oid_do_índice, número_da_coluna, impressão_alinhada)</code>	text	obtém o comando CREATE INDEX para o índice, ou a definição de apenas uma coluna do índice quando o <code>número_da_coluna</code> não for zero
<code>pg_get_triggerdef(oid_do_gatilho)</code>	text	obtém o comando CREATE [ CONSTRAINT ] TRIGGER para o gatilho
<code>pg_get_constraintdef(oid_da_restrição)</code>	text	obtém a definição da restrição
<code>pg_get_constraintdef(oid_da_restrição, impressão_alinhada)</code>	text	obtém a definição da restrição
<code>pg_get_expr(texto_da_expressão, oid_da_relação)</code>	text	forma interna descompilada da expressão, assumindo que todas as Vars na mesma se referem à relação indicada pelo segundo parâmetro

Nome	Tipo retornado	Descrição
<code>pg_get_expr(texto_da_expressão, oid_da_relação, impressão_alinhada)</code>	text	forma interna descompilada da expressão, assumindo que todas as Vars na mesma se referem à relação indicada pelo segundo parâmetro
<code>pg_get_userbyid(id_do_usuario)</code>	name	obtém o nome do usuário com o identificador fornecido
<code>pg_get_serial_sequence(nome_da_tabela, nome_da_coluna)</code>	text	obtém o nome da sequência utilizada pela coluna <code>serial</code> ou <code>bigserial</code>
<code>pg_tablespace_databases(oid_do_espaco_de_tabelas)</code>	setof oid	obtém o conjunto de OIDs de banco de dados que possuem objetos no espaço de tabelas
Notas: a. <code>deprecated</code> — Dito de um programa ou funcionalidade que é considerada em obsolescência e no processo de ter sua utilização gradualmente interrompida, geralmente em favor de uma determinada substituição. As funcionalidades em obsolescência podem infelizmente, demorar muitos anos para desaparecer. The Jargon File ( <a href="http://www.catb.org/~esr/jargon/html/D/deprecated.html">http://www.catb.org/~esr/jargon/html/D/deprecated.html</a> ) (N. do T.)		

A função `format_type` retorna o nome SQL do tipo de dado que é identificado pelo seu OID de tipo e, possivelmente, um modificador de tipo. Deve ser passado NULL para o modificador de tipo se não for conhecido nenhum modificador específico.

As funções `pg_get_viewdef`, `pg_get_ruledef`, `pg_get_indexdef`, `pg_get_triggerdef` e `pg_get_constraintdef` reconstroem, respectivamente, o comando de criação da visão, regra, índice, gatilho e restrição; deve ser observado que esta é uma reconstrução por descompilação, e não o texto original do comando. A função `pg_get_expr` descompila a forma interna de uma expressão individual, tal como o valor padrão para uma coluna. Pode ser útil ao examinar o conteúdo dos catálogos do sistema. A maioria destas funções estão presentes em duas formas, uma das quais pode, opcionalmente, gerar o resultado de forma “alinhada” (`pretty-print`). O formato alinhado é mais legível, mas o formato padrão tem mais probabilidade de ser interpretado da mesma maneira nas versões futuras do PostgreSQL; deve ser evitada a utilização do formato alinhado com a finalidade de geração de dumps. Passar `false` para o parâmetro de impressão alinhada produz o mesmo resultado da variante que não possui o parâmetro.

A função `pg_get_userbyid` extrai o nome do usuário a partir do número de identificação do usuário. A função `pg_get_serial_sequence` busca o nome da sequência associada à coluna `serial` ou `bigserial`. O nome está adequadamente formatado para ser passado para as funções de sequência (consulte a Seção 9.12). Retorna NULL se a coluna não possui uma sequência associada.

A função `pg_tablespace_databases` permite examinar a utilização de um espaço de tabelas. Retorna um conjunto de OIDs de bancos de dados que possuem objetos armazenados no espaço de tabelas. Se esta função retornar alguma linha, então o espaço de tabelas não está vazio e, portanto, não pode ser removido. Para mostrar os objetos presentes no espaço de tabelas, é necessário se conectar ao banco de dados identificado por `pg_tablespace_databases` e consultar seus catálogos `pg_class`.

As funções mostradas na Tabela 9-45 extraem os comentários previamente armazenados pelo comando `COMMENT`. Retorna o valor nulo se não puder ser encontrado um comentário correspondendo aos parâmetros especificados.

**Tabela 9-45. Funções de informação de comentário**

Nome	Tipo retornado	Descrição
<code>obj_description(oid_do_objeto, nome_do_catálogo)</code>	text	obtém o comentário para o objeto do banco de dados

Nome	Tipo retornado	Descrição
<code>obj_description(oid_do_objeto)</code>	text	obtem o comentário para o objeto do banco de dados ( <i>em obsolescência</i> )
<code>col_description(oid_da_tabela, column_number)</code>	text	obtem o comentário para a coluna da tabela

A forma da função `obj_description()` com dois parâmetros retorna o comentário para o objeto do banco de dados especificado pelo seu OID e pelo nome do catálogo do sistema que o contém. Por exemplo, `obj_description(123456, 'pg_class')` retorna o comentário para a tabela com OID 123456. A forma de `obj_description()` com um parâmetro requer apenas o OID do objeto. Está obsoleta, porque não existe garantia dos OIDs serem únicos entre diferentes catálogos do sistema; portanto, pode ser retornado um comentário errado.

A função `col_description()` retorna o comentário para a coluna da tabela especificada pelo OID da tabela, e pelo número da coluna. A função `obj_description()` não pode ser utilizada para colunas de tabela, porque as colunas não possuem OIDs próprios.

### 9.19.1. Diferenças entre o PostgreSQL, o Oracle, o SQL Server e o DB2

**Nota:** Seção escrita pelo tradutor, não fazendo parte do manual original.

Esta seção tem por finalidade mostrar, através de exemplos práticos, as diferenças entre as funções de informação do sistema mostradas acima e suas equivalentes do PostgreSQL, do Oracle, do SQL Server e do DB2.

#### Exemplo 9-19. Funções de informação do sistema

Abaixo estão mostrados exemplos comparando a utilização das funções de informação do sistema no PostgreSQL, no SQL Server, no Oracle e no DB2. Consulte também `SYS_CONTEXT` (<http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/functions122a.htm>).

PostgreSQL 8.0.0:

```
=> SELECT current_database();

current_database
-----
templatel
(1 linha)

=> SELECT current_schema();

current_schema
-----
public
(1 linha)

=> SELECT current_schemas(TRUE);

current_schemas
-----
{pg_catalog,public}
(1 linha)

=> SELECT CURRENT_USER AS usuario;

usuario
-----
postgres
(1 linha)
```

```
=> SELECT inet_client_addr();
```

```
inet_client_addr
-----
127.0.0.1
(1 linha)
```

```
=> template1=# SELECT inet_client_port();
```

```
inet_client_port
-----
1151
(1 linha)
```

```
=> SELECT inet_server_addr();
```

```
inet_server_addr
-----
127.0.0.1
(1 linha)
```

```
=> SELECT inet_server_port();
```

```
inet_server_port
-----
5432
(1 linha)
```

```
=> SELECT SESSION_USER as usuario;
```

```
usuario
-----
postgres
(1 linha)
```

```
=> SELECT USER as usuario;
```

```
usuario
-----
postgres
(1 linha)
```

```
=> SELECT version() AS versao;
```

```
versao
-----
PostgreSQL 8.0.0 on i686-pc-mingw32, compiled by GCC gcc.exe (GCC) 3.4.2 (mingw-special)
(1 linha)
```

SQL Server 2000:

```
SELECT db_name() AS current_database
```

```
current_database
-----
pubs
(1 row(s) affected)
```

```
SELECT CURRENT_USER AS usuario;
```

```
usuario
-----
dbo
(1 row(s) affected)
```

```
SELECT SESSION_USER as usuario
```

```
usuario
-----
dbo
(1 row(s) affected)
```

```
SELECT SYSTEM_USER as usuario
```

```
usuario
-----
halley
(1 row(s) affected)
```

```
SELECT host_name() AS nome_hospedeiro -- cliente
```

```
nome_hospedeiro
-----
BD01
(1 row(s) affected)
```

```
SELECT @@VERSION AS versao
```

```
versao
-----
Microsoft SQL Server 2000 - 8.00.760 (Intel X86)
    Dec 17 2002 14:22:05
    Copyright (c) 1988-2003 Microsoft Corporation
    Enterprise Edition on Windows NT 5.2 (Build 3790: )
(1 row(s) affected)
```

Oracle 10g:

```
SQL> SELECT sys_context ('USERENV', 'DB_NAME') FROM sys.dual;
```

```
SYS_CONTEXT('USERENV','DB_NAME')
-----
orcl
```

```
SQL> SELECT sys_context ('USERENV', 'CURRENT_SCHEMA') FROM sys.dual;
```

```
SYS_CONTEXT('USERENV','CURRENT_SCHEMA')
-----
SCOTT
```

```
SQL> SELECT sys_context ('USERENV', 'CURRENT_USER') FROM sys.dual;
```

```
SYS_CONTEXT('USERENV','CURRENT_USER')
-----
SCOTT
```

```
SQL> SELECT sys_context ('USERENV', 'HOST') FROM sys.dual;
```

```
SYS_CONTEXT('USERENV','HOST')
-----
CASA\HALLEY
```

```
SQL> SELECT sys_context ('USERENV', 'IP_ADDRESS') FROM sys.dual;
```

```
SYS_CONTEXT('USERENV','IP_ADDRESS')
-----
200.165.203.130
```



```

SQL> SELECT sys_context ('USERENV', 'LANGUAGE') FROM sys.dual;

SYS_CONTEXT('USERENV','LANGUAGE')
-----
AMERICAN_AMERICA.WE8ISO8859P1

SQL> SELECT sys_context ('USERENV', 'NETWORK_PROTOCOL') FROM sys.dual;

SYS_CONTEXT('USERENV','NETWORK_PROTOCOL')
-----
tcp

SQL> SELECT sys_context ('USERENV', 'NLS_DATE_FORMAT') FROM sys.dual;

SYS_CONTEXT('USERENV','NLS_DATE_FORMAT')
-----
DD-MON-RR

SQL> SELECT sys_context ('USERENV', 'OS_USER') FROM sys.dual;

SYS_CONTEXT('USERENV','OS_USER')
-----
HALLEY\Administrator

SQL> SELECT sys_context ('USERENV', 'SESSION_USER') FROM sys.dual;

SYS_CONTEXT('USERENV','SESSION_USER')
-----
SCOTT

SQL> SELECT banner FROM v$version;

BANNER
-----
Oracle Database 10g Enterprise Edition Release 10.1.0.2.0 - Prod
PL/SQL Release 10.1.0.2.0 - Production
CORE      10.1.0.2.0      Production
TNS for 32-bit Windows: Version 10.1.0.2.0 - Production
NLSRTL Version 10.1.0.2.0 - Production

DB2 8.1:

DB2SQL92> SELECT CURRENT_SCHEMA FROM sysibm.sysdummy1;

1
-----
DB2INST1

DB2SQL92> SELECT CURRENT_SERVER FROM sysibm.sysdummy1;

1
-----
SAMPLE

DB2SQL92> SELECT USER FROM sysibm.sysdummy1;

1
-----
DB2INST1

-- Linha de comando

```

```
$ db2level
```

```
DB21085I  A instância "db2inst1" utiliza "32" bits e o release de código do DB2
"SQL08020" com identificador de nível "03010106".
Os tokens informativos são "DB2 v8.1.0.64", "s040812", "MI00086" e FixPak "7".
O produto está instalado em "/opt/IBM/db2/V8.1".
```

## 9.20. Funções para administração do sistema

A Tabela 9-46 mostra as funções disponíveis para consultar e para alterar os parâmetros de configuração em tempo de execução.

**Tabela 9-46. Funções para definição de configuração**

Nome	Tipo retornado	Descrição
<code>current_setting(nome_da_definição)</code>	text	valor corrente da definição
<code>set_config(nome_da_definição, novo_valor, é_local)</code>	text	define o parâmetro e retorna o novo valor

A função `current_setting` retorna o valor corrente da definição `nome_da_definição`. Corresponde ao comando SQL `SHOW`. Por exemplo:

```
=> SELECT current_setting('datestyle');
```

```
current_setting
-----
ISO, DMY
(1 linha)
```

A função `set_config` define o parâmetro `nome_da_configuração` como `novo_valor`. Se o parâmetro `é_local` for `true`, então o novo valor se aplica somente à transação corrente. Se for desejado que o novo valor seja aplicado à sessão corrente, deve ser utilizado `false`. Esta função corresponde ao comando SQL `SET`. Por exemplo:

```
=> SELECT set_config('log_statement_stats', 'off', false);
```

```
set_config
-----
off
(1 linha)
```

A função mostrada na Tabela 9-47 envia sinais de controle para outros processos servidor. A utilização desta função é restrita aos superusuários.

**Tabela 9-47. Funções de sinais para o servidor**

Nome	Tipo retornado	Descrição
<code>pg_cancel_backend(pid)</code>	int	Cancela o comando corrente no servidor

Se for bem-sucedida a função retorna 1, caso contrário retorna 0. O ID do processo (`pid`) de um servidor ativo pode ser encontrado a partir da coluna `procpid` da visão `pg_stat_activity`, ou listando os processos do `postgres` no servidor através do comando do Unix `ps`.

As funções mostradas na Tabela 9-48 ajudam a realizar cópias de segurança *on-line*. A utilização destas funções é restrita aos superusuários.

**Tabela 9-48. Funções de controle de cópia de segurança**

Nome	Tipo retornado	Descrição
<code>pg_start_backup(texto_do_rótulo)</code>	text	configura para realizar cópia de segurança on-line
<code>pg_stop_backup()</code>	text	Termina a realização da cópia de segurança on-line

A função `pg_start_backup` recebe um único parâmetro que é um rótulo arbitrário definido pelo usuário para a cópia de segurança (habitualmente é o nome sob o qual o arquivo da cópia de segurança será armazenado). A função escreve o arquivo rótulo da cópia de segurança no diretório de dados do agrupamento de bancos de dados e, então, retorna o deslocamento no WAL do início da cópia de segurança como texto (o usuário não precisa prestar atenção a este valor do resultado, mas é fornecido para o caso de ser útil).

A função `pg_stop_backup` remove o arquivo rótulo criado pela função `pg_start_backup` e, em seu lugar, cria o arquivo de histórico de cópia de segurança na área de arquivamento do WAL. O arquivo de histórico inclui o rótulo fornecido para a função `pg_start_backup`, os deslocamentos inicial e final do WAL para a cópia de segurança, e o tempo de início e de fim da cópia de segurança. O valor retornado é o deslocamento no WAL do fim da cópia de segurança (que, novamente, pode ser pouco interessante).

Para obter detalhes sobre a utilização apropriada destas funções consulte a Seção 22.3.

## Notas

1. `bitwise` — um operador bit a bit trata seus operandos como um vetor de bits, em vez de como um único número. FOLDOC - Free On-Line Dictionary of Computing (<http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=bitwise>) (N. do T.)
2. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
3. C, POSIX, `pt_BR`, etc. (N. do T.)
4. `collation; collating sequence` — Um método para comparar duas cadeias de caracteres comparáveis. Todo conjunto de caracteres possui seu `collation` padrão. (Second Informal Review Draft) ISO/IEC 9075:1992, Database Language SQL- July 30, 1992. (N. do T.)
5. Exemplo escrito pelo tradutor, não fazendo parte do manual original. Fora Inês Pedrosa, escritora portuguesa, os demais nomes foram tirados de *Women of the Last Millennium* ([http://www2.xlibris.com/bookstore/book\\_excerpt.asp?bookid=801](http://www2.xlibris.com/bookstore/book_excerpt.asp?bookid=801)).
6. Exemplo escrito pelo tradutor, não fazendo parte do manual original. As informações usadas neste exemplo foram retiradas da página Comissão de Legislação Participativa da Câmara dos Deputados (<http://www2.camara.gov.br/conheca/ouvidoria/dicas/dicasLegislativo.html>). Outros exemplos de expressões regulares podem ser encontrados em Regular Expression Library (<http://www.regexlib.com/>), Server-Side Validations Using Regular Expressions (<http://www.15seconds.com/issue/010301.htm>) e Regular Expression HOWTO (<http://www.amk.ca/python/howto/regex/>).
7. Exemplo escrito pelo tradutor, baseado no exemplo do livro *PHP 5 Programação Poderosa* de Andi Gutmans, Stig Saether Bakken e Derik Rethans - Editora Alta Books - 2005 - pág. 187
8. Exemplo escrito pelo tradutor, utilizando os parâmetros do exemplo anterior.
9. 60 se estiverem implementados no sistema operacional os segundos intercalados (`leap seconds`).
10. Em certas ocasiões, o UTC é ajustado pela omissão de um segundo ou a inserção do “segundo intercalado” para manter sincronismo com o tempo sideral. Isto implica que às vezes, mas muito raramente, um determinado minuto contém exatamente 59, 61 ou 62 segundos. Se a implementação do SQL suporta os segundos intercalados, e as consequências deste suporte para aritmética de data e intervalo, é definido pela implementação. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
11. O Brasil está a oeste da UTC (ocidente). O horário de Brasília normal corresponde ao GMT-3, e durante o horário de verão corresponde ao GMT-2. (N. do T.)

12. A função CURRENT\_TIMESTAMP é definida no padrão SQL possuindo o formato <current timestamp value function> ::= CURRENT\_TIMESTAMP [ <left paren> <timestamp precision> <right paren> ] (Second Informal Review Draft) ISO/IEC 9075:1992, Database Language SQL- July 30, 1992. (N. do T.)
13. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
14. A tabela de associação pode ser vista em <http://standards.ieee.org/regauth/oui/oui.txt> (N. do T.)
15. SQL Server 2000 — Aceita tanto a forma geral quanto a forma simplificada da expressão CASE. (N. do T.)
16. Oracle 10g — Aceita tanto a forma geral quanto a forma simplificada da expressão CASE. (N. do T.)
17. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
18. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
19. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
20. SQL Server 2000 — SOME | ANY comparam um valor escalar com o conjunto de valores de uma única coluna. Sintaxe: *expressão\_escalar* { = | < | > | ! = | > | > = | ! > | < | < = | ! < } { SOME | ANY } ( *subconsulta* ). A subconsulta possui o conjunto de resultados de uma coluna, e o mesmo tipo de dado da expressão escalar. SOME e ANY retornam verdade quando a comparação especificada é verdade para qualquer par (*expressão\_escalar*, x), onde x é um valor do conjunto de uma única coluna. Senão retorna falso. SOME | ANY ([http://msdn.microsoft.com/library/en-us/tsqlref/ts\\_setu-sus\\_17jt.asp](http://msdn.microsoft.com/library/en-us/tsqlref/ts_setu-sus_17jt.asp)) (N. do T.)
21. Exemplo escrito pelo tradutor, não fazendo parte do manual original.

# Capítulo 10. Conversão de tipo

Os comandos SQL podem, intencionalmente ou não, usar tipos de dado diferentes na mesma expressão. O PostgreSQL possui muitas funcionalidades para processar expressões com mistura de tipos.

Em muitos casos não há necessidade do usuário compreender os detalhes do mecanismo de conversão de tipo. Entretanto, as conversões implícitas feitas pelo PostgreSQL podem afetar o resultado do comando. Quando for necessário, os resultados podem ser personalizados utilizando uma conversão de tipo *explícita*.

Este capítulo apresenta os mecanismos e as convenções de conversão de tipo de dado do PostgreSQL. Consulte as seções relevantes no Capítulo 8 e no Capítulo 9 para obter informações adicionais sobre tipos de dado específicos, e funções e operadores permitidos, respectivamente.

## 10.1. Visão geral

A linguagem SQL é uma linguagem fortemente tipada, ou seja, todo item de dado possui um tipo de dado associado que determina seu comportamento e a utilização permitida. O PostgreSQL possui um sistema de tipo de dado extensivo, muito mais geral e flexível do que o de outras implementações do SQL. Por isso, a maior parte do comportamento de conversão de tipo de dado do PostgreSQL é governado por regras gerais, em vez de heurísticas <sup>1</sup> *ad hoc* <sup>2</sup>, permitindo, assim, expressões com tipos diferentes terem significado mesmo com tipos definidos pelo usuário.

O rastreador/analizador (*scanner/parser*) do PostgreSQL divide os elementos léxicos em apenas cinco categorias fundamentais: inteiros, números não inteiros, cadeias de caracteres, identificadores e palavras chave. As constantes dos tipos não numéricos são, em sua maioria, classificadas inicialmente como cadeias de caracteres. A definição da linguagem SQL permite especificar o nome do tipo juntamente com a cadeia de caracteres, e este mecanismo pode ser utilizado no PostgreSQL para colocar o analisador no caminho correto. Por exemplo, a consulta

```
=> SELECT text 'Origem' AS "local", point '(0,0)' AS "valor";
```

```
local  | valor
-----+-----
Origem | (0,0)
(1 linha)
```

possui duas constantes literais, dos tipos `text` e `point`. Se o tipo do literal cadeia de caracteres não for especificado, inicialmente é atribuído o tipo provisório `unknown` (desconhecido), a ser determinado posteriormente nos estágios descritos abaixo.

Existem quatro construções SQL fundamentais que requerem regras de conversão de tipo distintas no analisador do PostgreSQL:

### Chamadas de função

Grande parte do sistema de tipo do PostgreSQL é construído em torno de um amplo conjunto de funções. As funções podem possuir um ou mais argumentos. Como o PostgreSQL permite a sobrecarga de funções, o nome da função, por si só, não identifica unicamente a função a ser chamada; o analisador deve selecionar a função correta baseando-se nos tipos de dado dos argumentos fornecidos.

### Operadores

O PostgreSQL permite expressões com operadores unários (um só argumento) de prefixo e de sufixo, assim como operadores binários (dois argumentos). Assim como as funções, os operadores podem ser sobrecarregados e, portanto, existe o mesmo problema para selecionar o operador correto.

### Armazenamento do valor

Os comandos SQL `INSERT` e `UPDATE` colocam os resultados das expressões em tabelas. As expressões nestes comandos devem corresponder aos tipos de dado das colunas de destino, ou talvez serem convertidas para estes tipos de dado.

## Construções UNION, CASE e ARRAY

Como os resultados de todas as cláusulas `SELECT` de uma declaração envolvendo união devem aparecer em um único conjunto de colunas, deve ser feita a correspondência entre os tipos de dado dos resultados de todas as cláusulas `SELECT` e a conversão em um conjunto uniforme. Do mesmo modo, os resultados das expressões da construção `CASE` devem ser todos convertidos em um tipo de dado comum, para que a expressão `CASE` tenha, como um todo, um tipo de dado de saída conhecido. O mesmo se aplica às construções `ARRAY`.

Os catálogos do sistema armazenam informações sobre que conversões entre tipos de dado, chamadas de *casts*, são válidas, e como realizar estas conversões. Novas conversões podem ser adicionadas pelo usuário através do comando `CREATE CAST` (Geralmente isto é feito junto com a definição de novos tipos de dado. O conjunto de conversões entre os tipos nativos foi cuidadosamente elaborado, sendo melhor não alterá-lo).

É fornecida no analisador uma heurística adicional para permitir estimar melhor o comportamento apropriado para os tipos do padrão SQL. Existem diversas *categorias de tipo* básicas definidas: `boolean`, `numeric`, `string`, `bitstring`, `datetime`, `timespan`, `geometric`, `network` e a definida pelo usuário. Cada categoria, com exceção da definida pelo usuário, possui um ou mais *tipo preferido*, selecionado preferencialmente quando há ambiguidade. Na categoria definida pelo usuário, cada tipo é o seu próprio tipo preferido. As expressões ambíguas (àquelas com várias soluções de análise candidatas) geralmente podem, portanto, serem resolvidas quando existem vários tipos nativos possíveis, mas geram erro quando existem várias escolhas para tipos definidos pelo usuário.

Todas as regras de conversão de tipo foram projetadas com vários princípios em mente:

- As conversões implícitas nunca devem produzir resultados surpreendentes ou imprevisíveis.
- Tipos definidos pelo usuário, para os quais o analisador não possui nenhum conhecimento *a priori*, devem estar “acima” na hierarquia de tipo. Nas expressões com tipos mistos, os tipos nativos devem sempre ser convertidos no tipo definido pelo usuário (obviamente, apenas se a conversão for necessária).
- Tipos definidos pelo usuário não se relacionam. Atualmente o PostgreSQL não dispõe de informações sobre o relacionamento entre tipos, além das heurísticas codificadas para os tipos nativos e relacionamentos implícitos baseado nas funções e conversões disponíveis.
- Não deve haver nenhum trabalho extra do analisador ou do executor se o comando não necessitar de conversão de tipo implícita, ou seja, se o comando estiver bem formulado e os tipos já se correspondem, então o comando deve prosseguir sem despendar tempo adicional no analisador, e sem introduzir chamadas de conversão implícita desnecessárias no comando.

Além disso, se o comando geralmente requer uma conversão implícita para a função, e se o usuário definir uma nova função com tipos corretos para os argumentos, então o analisador deve usar esta nova função, não fazendo mais a conversão implícita utilizando a função antiga.

## 10.2. Operadores

O operador específico a ser usado na chamada de operador é determinado pelo procedimento mostrado abaixo. Deve ser observado que este procedimento é afetado indiretamente pela precedência dos operadores envolvidos. Consulte a Seção 4.1.6 para obter informações adicionais.

### Resolução do tipo em operando

1. Selecionar no catálogo do sistema `pg_operator` os operadores a serem considerados. Se for utilizado um nome de operador não qualificado (o caso usual), os operadores a serem considerados são aqueles com nome e número de argumentos corretos, visíveis no caminho de procura corrente (consulte a Seção 5.8.3). Se for utilizado um nome de operador qualificado, somente são considerados os operadores no esquema especificado.
  - a. Se forem encontrados no caminho de procura vários operadores com argumentos do mesmo tipo, somente é considerado aquele que aparece primeiro no caminho. Porém, os operadores com argumentos de tipos diferentes são considerados em pé de igualdade, não importando a posição no caminho de procura.
2. Verificar se algum operador aceita exatamente os mesmos tipos de dado dos argumentos da entrada. Caso exista (só pode haver uma correspondência exata no conjunto de operadores considerados), este é usado.

- a. Se um dos argumentos da chamada do operador binário for do tipo `unknown` (desconhecido), então assumir que seja do mesmo tipo do outro argumento nesta verificação. Outros casos envolvendo o tipo `unknown` nunca encontram correspondência nesta etapa.
3. Procurar pela melhor correspondência.
- a. Desconsiderar os operadores candidatos para os quais os tipos da entrada não correspondem, e não podem ser convertidos (utilizando uma conversão implícita) para corresponder. Para esta finalidade é assumido que os literais do tipo `unknown` podem ser convertidos em qualquer tipo. Se permanecer apenas um operador candidato, então este é usado; senão continuar na próxima etapa.
  - b. Examinar todos os operadores candidatos, e manter aqueles com mais correspondências exatas com os tipos da entrada (Os domínios são considerados idênticos aos seus tipos base para esta finalidade). Manter todos os candidatos se nenhum possuir alguma correspondência exata. Se apenas um candidato permanecer, este é usado; senão continuar na próxima etapa.
  - c. Examinar todos os operadores candidatos, e manter aqueles que aceitam os tipos preferidos (da categoria de tipo do tipo de dado da entrada) em mais posições onde a conversão de tipo será necessária. Manter todos os candidatos se nenhum aceitar os tipos preferidos. Se apenas um operador candidato permanecer, este é usado; senão continuar na próxima etapa.
  - d. Se algum dos argumentos de entrada for do tipo “unknown”, verificar as categorias de tipo aceitas nesta posição do argumento pelos candidatos remanescentes. Em cada posição, selecionar a categoria `string` se qualquer um dos candidatos aceitar esta categoria (este favorecimento em relação à cadeia de caracteres é apropriado, porque um literal de tipo desconhecido se parece com uma cadeia de caracteres). Senão, se todos os candidatos remanescentes aceitarem a mesma categoria de tipo, selecionar esta categoria; senão falhar, porque a escolha correta não pode ser deduzida sem informações adicionais. Agora, rejeitar os operadores candidatos que não aceitam a categoria de tipo selecionada; além disso, se algum operador candidato aceitar o tipo preferido em uma determinada posição do argumento, rejeitar os candidatos que aceitam tipos não preferidos para este argumento.
  - e. Se permanecer apenas um operador candidato, este é usado; Se não permanecer nenhum candidato, ou permanecer mais de um candidato, então falhar.

Seguem alguns exemplos.

#### Exemplo 10-1. Resolução do tipo em operador de exponenciação

Existe apenas um operador de exponenciação definido no catálogo, e recebe argumentos do tipo `double precision`. O rastreador atribui o tipo inicial `integer` aos os dois argumentos desta expressão de consulta:

```
=> SELECT 2 ^ 3 AS "exp";
```

```
exp
----
  8
(1 linha)
```

Portanto, o analisador faz uma conversão de tipo nos dois operandos e a consulta fica equivalente a

```
=> SELECT CAST(2 AS double precision) ^ CAST(3 AS double precision) AS "exp";
```

#### Exemplo 10-2. Resolução do tipo em operador de concatenação de cadeia de caracteres

Uma sintaxe estilo cadeia de caracteres é utilizada para trabalhar com tipos cadeias de caracteres, assim como para trabalhar com tipos de extensão complexa. Cadeias de caracteres de tipo não especificado se correspondem com praticamente todos os operadores candidatos.

Um exemplo com um argumento não especificado:

```
=> SELECT text 'abc' || 'def' AS "texto e desconhecido";
```

```
texto e desconhecido
-----
abcdef
(1 linha)
```

Neste caso o analisador procura pela existência de algum operador recebendo o tipo `text` nos dois argumentos. Uma vez que existe, assume que o segundo argumento deve ser interpretado como sendo do tipo `text`.

Concatenação de tipos não especificados:

```
=> SELECT 'abc' || 'def' AS "não especificado";
```

```
 não especificado
-----
 abcdef
(1 linha)
```

Neste caso não existe nenhuma pista inicial do tipo a ser usado, porque não foi especificado nenhum tipo na consulta. Portanto, o analisador procura todos os operadores candidatos, e descobre que existem candidatos aceitando tanto cadeia de caracteres quanto cadeia de bits como entrada. Como a categoria cadeia de caracteres é a preferida quando está disponível, esta categoria é selecionada e, depois, é usado o tipo preferido para cadeia de caracteres, `text`, como o tipo específico para solucionar os literais de tipo desconhecido.

### Exemplo 10-3. Resolução do tipo em operador de valor absoluto e negação

O catálogo de operadores do PostgreSQL possui várias entradas para o operador de prefixo `@`, todas implementando operações de valor absoluto para vários tipos de dado numéricos. Uma destas entradas é para o tipo `float8`, que é o tipo preferido da categoria numérica. Portanto, o PostgreSQL usa esta entrada quando na presença de uma entrada não numérica:

```
=> SELECT @ '-4.5' AS "abs";
```

```
 abs
----
 4.5
(1 linha)
```

Aqui o sistema realiza uma conversão implícita de `text` para `float8` antes de aplicar o operador escolhido. Pode ser verificado que foi utilizado `float8`, e não algum outro tipo, escrevendo-se:

```
=> SELECT @ '-4.5e500' AS "abs";
```

```
ERRO:  "-4.5e500" está fora da faixa para o tipo double precision
```

Por outro lado, o operador de prefixo `~` (negação bit-a-bit) é definido apenas para tipos de dado inteiros, e não para `float8`. Portanto, se tentarmos algo semelhante usando `~`, resulta em:

```
=> SELECT ~ '20' AS "negação";
```

```
ERRO:  operador não é único: ~ "unknown"
DICA:  Não foi possível escolher um operador candidato melhor.
       Pode ser necessário adicionar uma conversão de tipo explícita.
```

Isto acontece porque o sistema não pode decidir qual dos vários operadores `~` possíveis deve ser o preferido. Pode ser dada uma ajuda usando uma conversão explícita:

```
=> SELECT ~ CAST('20' AS int8) AS "negação";
```

```
 negação
-----
      -21
(1 linha)
```

## 10.3. Funções

A função específica a ser utilizada em uma chamada de função é determinada de acordo com os seguintes passos.

### Resolução do tipo em função

1. Selecionar no catálogo do sistema `pg_proc` as funções a serem consideradas. Se for utilizado um nome de função não qualificado, as funções consideradas são aquelas com nome e número de argumentos corretos, visíveis no caminho de procura corrente (consulte a Seção 5.8.3). Se for fornecido um nome de função qualificado, somente são consideradas as funções no esquema especificado.



- a. Se forem encontradas no caminho de procura várias funções com argumentos do mesmo tipo, somente é considerada àquela que aparece primeiro no caminho. Mas as funções com argumentos de tipos diferentes são consideradas em pé de igualdade, não importando a posição no caminho de procura.
2. Verificar se alguma função aceita exatamente os mesmos tipos de dado dos argumentos de entrada. Caso exista (só pode haver uma correspondência exata no conjunto de funções consideradas), esta é usada. Os casos envolvendo o tipo `unknown` nunca encontram correspondência nesta etapa.
3. Se não for encontrada nenhuma correspondência exata, verificar se a chamada de função parece ser uma solicitação trivial de conversão de tipo. Isto acontece quando a chamada de função possui apenas um argumento, e o nome da função é o mesmo nome (interno) de algum tipo de dado. Além disso, o argumento da função deve ser um literal de tipo desconhecido, ou um tipo binariamente compatível com o tipo de dado do nome da função. Quando estas condições são satisfeitas, o argumento da função é convertido no tipo de dado do nome da função sem uma chamada real de função.
4. Procurar pela melhor correspondência.
  - a. Desprezar as funções candidatas para as quais os tipos da entrada não correspondem, e nem podem ser convertidos (utilizando uma conversão implícita) para corresponder. Para esta finalidade é assumido que os literais do tipo `unknown` podem ser convertidos em qualquer tipo. Se permanecer apenas uma função candidata, então esta é usada; senão continuar na próxima etapa.
  - b. Examinar todas as funções candidatas, e manter aquelas com mais correspondências exatas com os tipos da entrada (Para esta finalidade os domínios são considerados idênticos aos seus tipos base). Manter todas as funções candidatas se nenhuma possuir alguma correspondência exata. Se permanecer apenas uma função candidata, então esta é usada; senão continuar na próxima etapa.
  - c. Examinar todas as funções candidatas, e manter aquelas que aceitam os tipos preferidos (da categoria de tipo do tipo de dado de entrada) em mais posições onde a conversão de tipo será necessária. Manter todas as candidatas se nenhuma aceitar o tipo preferido. Se permanecer apenas uma função candidata, esta é usada; senão continuar na próxima etapa.
  - d. Se algum dos argumentos de entrada for do tipo “unknown”, verificar as categorias de tipo aceitas nesta posição do argumento pelas funções candidatas remanescentes. Em cada posição, selecionar a categoria `string` se qualquer uma das candidatas aceitar esta categoria (este favorecimento em relação à cadeia de caracteres é apropriado, porque um literal de tipo desconhecido se parece com uma cadeia de caracteres). Senão, se todas as candidatas remanescentes aceitam a mesma categoria de tipo, selecionar esta categoria; senão falhar, porque a escolha correta não pode ser deduzida sem informações adicionais. Rejeitar agora as funções candidatas que não aceitam a categoria de tipo selecionada; além disso, se alguma função candidata aceitar o tipo preferido em uma dada posição do argumento, rejeitar as candidatas que aceitam tipos não preferidos para este argumento.
  - e. Se permanecer apenas uma função candidata, este é usada; Se não permanecer nenhuma função candidata, ou se permanecer mais de uma candidata, então falhar.

Deve ser observado que as regras da “melhor correspondência” são idênticas para a resolução do tipo em operador e função. Seguem alguns exemplos.

#### Exemplo 10-4. Resolução do tipo do argumento em função de arredondamento

Existe apenas uma função `round` com dois argumentos (O primeiro é `numeric` e o segundo é `integer`). Portanto, a consulta abaixo converte automaticamente o primeiro argumento do tipo `integer` para `numeric`:

```
=> SELECT round(4, 4);
```

```
round
-----
4.0000
(1 linha)
```

Na verdade esta consulta é convertida pelo analisador em

```
=> SELECT round(CAST (4 AS numeric), 4);
```

Uma vez que inicialmente é atribuído o tipo `numeric` às constantes numéricas com ponto decimal, a consulta abaixo não necessita de conversão de tipo podendo, portanto, ser ligeiramente mais eficiente:

```
=> SELECT round(4.0, 4);
```

#### Exemplo 10-5. Resolução do tipo em função de subcadeia de caracteres

Existem diversas funções `substr`, uma das quais aceita os tipos `text` e `integer`. Se esta função for chamada com uma constante cadeia de caracteres de tipo não especificado, o sistema escolhe a função candidata que aceita o argumento da categoria preferida para `string` (que é o tipo `text`).

```
=> SELECT substr('1234', 3);
```

```
substr
-----
      34
(1 linha)
```

Se a cadeia de caracteres for declarada como sendo do tipo `varchar`, o que pode ser o caso se vier de uma tabela, então o analisador tenta converter para torná-la do tipo `text`:

```
=> SELECT substr(varchar '1234', 3);
```

```
substr
-----
      34
(1 linha)
```

Esta consulta é transformada pelo analisador para se tornar efetivamente:

```
=> SELECT substr(CAST (varchar '1234' AS text), 3);
```

**Nota:** O analisador descobre no catálogo `pg_cast` que os tipos `text` e `varchar` são binariamente compatíveis, significando que um pode ser passado para uma função que aceita o outro sem realizar qualquer conversão física. Portanto, neste caso, não é realmente inserida nenhuma chamada de conversão de tipo explícita.

E, se a função for chamada com um argumento do tipo `integer`, o analisador tentará convertê-lo em `text`:

```
=> SELECT substr(1234, 3);
```

```
substr
-----
      34
(1 linha)
```

Na verdade é executado como:

```
=> SELECT substr(CAST (1234 AS text), 3);
```

Esta transformação automática pode ser feita, porque existe uma conversão implícita de `integer` para `text` que pode ser chamada.

## 10.4. Armazenamento de valor

Os valores a serem inseridos na tabela são convertidos no tipo de dado da coluna de destino de acordo com as seguintes etapas.

### Conversão de tipo para armazenamento de valor

1. Verificar a correspondência exata com o destino.
2. Senão, tentar converter a expressão no tipo de dado de destino. Isto será bem-sucedido se houver uma conversão registrada entre os dois tipos. Se a expressão for um literal de tipo desconhecido, o conteúdo do literal cadeia de caracteres será enviado para a rotina de conversão de entrada do tipo de destino.
3. Verificar se existe uma conversão de tamanho para o tipo de destino. Uma conversão de tamanho é uma conversão do tipo para o próprio tipo. Se for encontrada alguma no catálogo `pg_cast` aplicá-la à expressão antes de armazenar na coluna de destino. A função que implementa este tipo de conversão sempre aceita um parâmetro adicional do tipo `integer`, que recebe o comprimento declarado da coluna de destino (na verdade, seu valor `atttypmod`; a interpretação de `atttypmod` varia entre tipos de dado diferentes). A função de conversão é responsável por aplicar toda semântica dependente do comprimento, tal como verificação do tamanho ou truncamento.

**Exemplo 10-6. Conversão de tipo no armazenamento de character**

Para uma coluna de destino declarada como `character(20)`, a seguinte declaração garante que o valor armazenado terá o tamanho correto:

```
=> CREATE TABLE vv (v character(20));
=> INSERT INTO vv SELECT 'abc' || 'def';
=> SELECT v, length(v) FROM vv;
```

```

      v          | length
-----+-----
 abcdef         |      20
(1 linha)
```

O que acontece realmente aqui, é que os dois literais desconhecidos são resolvidos como `text` por padrão, permitindo que o operador `||` seja resolvido como concatenação de `text`. Depois, o resultado `text` do operador é convertido em `bpchar` (“caractere completado com brancos”, ou “blank-padded char”, que é o nome interno do tipo de dado `character`) para corresponder com o tipo da coluna de destino (Uma vez que os tipos `text` e `bpchar` são binariamente compatíveis, esta conversão não insere nenhuma chamada real de função). Por fim, a função de tamanho `bpchar(bpchar, integer)` é encontrada no catálogo do sistema, e aplicada ao resultado do operador e comprimento da coluna armazenada. Esta função específica do tipo realiza a verificação do comprimento requerido, e adiciona espaços para completar.

**10.5. Construções UNION, CASE e ARRAY**

As construções `UNION` do SQL precisam unir tipos, que podem não ser semelhantes, para que se tornem um único conjunto de resultados. O algoritmo de resolução é aplicado separadamente a cada coluna de saída da consulta união. As construções `INTERSECT` e `EXCEPT` resolvem tipos não semelhantes do mesmo modo que `UNION`. As construções `CASE` e `ARRAY` utilizam um algoritmo idêntico para fazer a correspondência das expressões componentes e selecionar o tipo de dado do resultado.

**Resolução do tipo em UNION, CASE e ARRAY**

1. Se todas as entradas forem do tipo `unknown`, é resolvido como sendo do tipo `text` (o tipo preferido da categoria cadeia de caracteres). Senão, ignorar as entradas `unknown` ao escolher o tipo do resultado.
2. Se as entradas não-desconhecidas não forem todas da mesma categoria de tipo, falhar.
3. Escolher o primeiro tipo de entrada não-desconhecido que for o tipo preferido nesta categoria, ou que permita todas as entradas não-desconhecidas serem convertidas implicitamente no mesmo.
4. Converter todas as entradas no tipo selecionado.

Seguem alguns exemplos.

**Exemplo 10-7. Resolução do tipo com tipos subespecificados em uma união**

```
=> SELECT text 'a' AS "texto" UNION SELECT 'b';
```

```

 texto
-----
 a
 b
(2 linhas)
```

Neste caso, o literal de tipo desconhecido `'b'` é resolvido como sendo do tipo `text`.

**Exemplo 10-8. Resolução do tipo em uma união simples**

```
=> SELECT 1.2 AS "numérico" UNION SELECT 1;
```

```

 numérico
-----
      1
     1.2
(2 linhas)
```

O literal `1.2` é do tipo `numeric`, e o valor inteiro `1` pode ser convertido implicitamente em `numeric`, portanto este tipo é utilizado.

#### Exemplo 10-9. Resolução do tipo em uma união transposta

```
=> SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);
```

```
real
-----
    1
   2.2
(2 linhas)
```

Neste caso, como o tipo `real` não pode ser convertido implicitamente em `integer`, mas `integer` pode ser implicitamente convertido em `real`, o tipo do resultado da união é resolvido como `real`.

## Notas

heurística — conjunto de regras e métodos que conduzem à descoberta, à invenção e à resolução de problemas. Novo Dicionário Aurélio da Língua Portuguesa. (N. do T.)

ad hoc — para isso, para esse caso. Novo Dicionário Aurélio da Língua Portuguesa. (N. do T.)

# Capítulo 11. Índices

Os índices são um modo comum de melhorar o desempenho do banco de dados. O índice permite ao servidor de banco de dados encontrar e trazer linhas específicas muito mais rápido do que faria sem o índice. Entretanto, os índices também produzem trabalho adicional para o sistema de banco de dados como um todo devendo, portanto, serem utilizados com sensatez.

## 11.1. Introdução

Suponha a existência de uma tabela como:

```
CREATE TABLE testel (  
    id          integer,  
    conteudo    varchar  
);
```

e um aplicativo requerendo muitas consultas da forma:

```
SELECT conteudo FROM testel WHERE id = constante;
```

Sem preparo prévio, o sistema teria que varrer toda a tabela `testel`, linha por linha, para encontrar todas as entradas correspondentes. Havendo muitas linhas em `testel`, e somente poucas linhas (talvez somente uma ou nenhuma) retornadas pela consulta, então este método é claramente ineficiente. Porém, se o sistema fosse instruído para manter um índice para a coluna `id`, então poderia ser utilizado um método mais eficiente para localizar as linhas correspondentes. Por exemplo, só necessitaria percorrer uns poucos níveis dentro da árvore de procura.

Uma abordagem semelhante é utilizada pela maioria dos livros, fora os de ficção: os termos e os conceitos procurados freqüentemente pelos leitores são reunidos em um índice alfabético colocado no final do livro. O leitor interessado pode percorrer o índice rapidamente e ir direto para a página desejada, em vez de ter que ler o livro por inteiro em busca do que está procurando. Assim como é tarefa do autor prever os itens que os leitores mais provavelmente vão procurar, é tarefa do programador de banco de dados prever quais índices trarão benefícios.

Pode ser utilizado o seguinte comando para criar um índice na coluna `id`:

```
CREATE INDEX idx_testel_id ON testel (id);
```

O nome `idx_testel_id` pode ser escolhido livremente, mas deve ser usado algo que permita lembrar mais tarde para que serve o índice.

Para remover um índice é utilizado o comando `DROP INDEX`. Os índices podem ser adicionados ou removidos das tabelas a qualquer instante.

Após o índice ser criado, não é necessária mais nenhuma intervenção adicional: o sistema atualiza o índice quando a tabela é modificada, e utiliza o índice nas consultas quando julgar mais eficiente que a varredura seqüencial da tabela. Porém, talvez seja necessário executar regularmente o comando `ANALYZE` para atualizar as estatísticas, para permitir que o planejador de comandos tome as decisões corretas. Consulte o Capítulo 13 para obter informações sobre como descobrir se o índice está sendo utilizado; e quando e porque o planejador pode decidir *não* utilizar um índice.

Os índices também podem beneficiar os comandos de atualização (`UPDATE`) e de exclusão (`DELETE`) com condição de procura. Além disso, os índices também podem ser utilizados em consultas com junção. Portanto, um índice definido em uma coluna que faça parte da condição de junção pode acelerar, significativamente, a consulta.

Quando um índice é criado, o sistema precisa mantê-lo sincronizado com a tabela. Isto adiciona um trabalho extra para as operações de manipulação de dados. Portanto, os índices não essenciais ou não utilizados devem ser removidos. Deve ser observado que uma consulta ou um comando de manipulação de dados pode utilizar, no máximo, um índice por tabela.

## 11.2. Tipos de índice

O PostgreSQL disponibiliza vários tipos de índice: B-tree (árvore B), R-tree (árvore R), Hash <sup>1</sup> e GiST. Cada tipo de índice utiliza um algoritmo diferente, mais apropriado para tipos diferentes de consulta. Por padrão, o comando `CREATE INDEX` cria um índice B-tree, adequado para a maioria das situações comuns.

Os índices B-tree podem tratar consultas de igualdade e de faixa, em dados que podem ser classificados em alguma ordem. Em particular, o planejador de comandos do PostgreSQL leva em consideração utilizar um índice B-tree sempre que uma coluna indexada está envolvida em uma comparação utilizando um dos seguintes operadores:

```
<
<=
=
>=
>
```

As construções equivalentes a combinações destes operadores, tais como `BETWEEN` e `IN`, também podem ser implementadas com procura de índice B-tree (Mas deve ser observado que `IS NULL` não é equivalente a `=` e não é indexável).

O otimizador também pode utilizar um índice B-tree nos comandos envolvendo os operadores de correspondência com padrão `LIKE`, `ILIKE`, `~` e `~*`, se o padrão estiver ancorado ao início da cadeia de caracteres como, por exemplo, em `col LIKE 'foo%'` ou `col ~ '^foo'`, mas não em `col LIKE '%bar'`. Entretanto, se o servidor não utilizar o idioma C, será necessário criar um índice com uma classe de operadores especial, para dar suporte a indexação de consultas com correspondência com padrão. Consulte a Seção 11.6 adiante.

A consulta abaixo mostra o idioma. <sup>2</sup>

```
=> \pset title Idioma
=> SELECT name, setting FROM pg_settings WHERE name LIKE 'lc%';
```

Idioma	
name	setting
lc_collate	pt_BR.iso88591
lc_ctype	pt_BR.iso88591
lc_messages	pt_BR.iso88591
lc_monetary	pt_BR.iso88591
lc_numeric	pt_BR.iso88591
lc_time	pt_BR.iso88591

(6 linhas)

Os índices R-tree são adequados para consultas a dados espaciais. Para criar um índice R-tree deve ser utilizado um comando da forma:

```
CREATE INDEX nome ON tabela USING RTREE (coluna);
```

O planejador de comandos do PostgreSQL considera utilizar um índice R-tree sempre que a coluna indexada está envolvida em uma comparação utilizando um dos seguintes operadores:

```
<<
<
<>
>>
@
~=
&&
```

(Consulte a Seção 9.10 para conhecer o significado destes operadores).

Os índices hash podem tratar apenas comparações de igualdade simples. O planejador de comandos do PostgreSQL considera utilizar um índice hash sempre que a coluna indexada está envolvida em uma comparação utilizando o operador `=`. O seguinte comando é utilizado para criar um índice hash:

```
CREATE INDEX nome ON tabela USING HASH (coluna);
```

**Nota:** Os testes mostraram que os índices `hash` do PostgreSQL não têm desempenho melhor do que os índices `B-tree`, e que o tamanho e o tempo de construção dos índices `hash` são muito piores. Por estas razões, desencoraja-se a utilização dos índices `hash`.

Os índices `GiST` não são um único tipo de índice, mas em vez disto uma infraestrutura dentro da qual podem ser implementadas muitas estratégias de indexação diferentes. Assim sendo, os operadores em particular com os quais o índice `GiST` pode ser utilizado variam dependendo da estratégia de indexação (a *classe de operadores*). Para obter mais informações consulte a Capítulo 48.

O método de índice `B-tree` é uma implementação das árvores B de alta-simultaneidade de Lehman-Yao. O método de índice `R-tree` implementa árvores R padrão utilizando o algoritmo de partição quadrática de Guttman. O método de índice `hash` é uma implementação do `hashing` linear de Litwin. São mencionados os algoritmos utilizados somente para indicar que todos estes métodos de índice são inteiramente dinâmicos, não necessitando de otimização periódica (como é o caso, por exemplo, dos métodos de acesso `hash` estáticos).

### 11.3. Índices com várias colunas

Pode ser definido um índice contendo mais de uma coluna. Por exemplo, se existir uma tabela como:

```
CREATE TABLE teste2 (
    principal int,
    secundario int,
    nome      varchar
);
```

(Digamos que seja armazenado no banco de dados o diretório `/dev...`) e freqüentemente sejam feitas consultas como

```
SELECT nome
FROM   teste2
WHERE  principal = constante AND secundario = constante;
```

então é apropriado definir um índice contendo as colunas `principal` e `secundario` como, por exemplo,

```
CREATE INDEX idx_teste2_princ_sec ON teste2 (principal, secundario);
```

Atualmente, somente as implementações de `B-tree` e `GiST` suportam índices com várias colunas. Podem ser especificadas até 32 colunas (Este limite pode ser alterado durante a geração do PostgreSQL; consulte o arquivo `pg_config_manual.h`).

O planejador de comandos pode utilizar um índice com várias colunas, para comandos envolvendo a coluna mais à esquerda na definição do índice mais qualquer número de colunas listadas à sua direita, sem omissões. Por exemplo, um índice contendo (`a`, `b`, `c`) pode ser utilizado em comandos envolvendo todas as colunas `a`, `b` e `c`, ou em comandos envolvendo `a` e `b`, ou em comandos envolvendo apenas `a`, mas não em outras combinações (Em um comando envolvendo `a` e `c`, o planejador pode decidir utilizar o índice apenas para `a`, tratando `c` como uma coluna comum não indexada). Obviamente, cada coluna deve ser usada com os operadores apropriados para o tipo do índice; as cláusulas envolvendo outros operadores não são consideradas.

Os índices com várias colunas só podem ser utilizados se as cláusulas envolvendo as colunas indexadas forem ligadas por `AND`. Por exemplo,

```
SELECT nome
FROM   teste2
WHERE  principal = constante OR secundario = constante;
```

não pode utilizar o índice `idx_teste2_princ_sec` definido acima para procurar pelas duas colunas (Entretanto, pode ser utilizado para procurar apenas a coluna principal).

Os índices com várias colunas devem ser usados com moderação. Na maioria das vezes, um índice contendo apenas uma coluna é suficiente, economizando espaço e tempo. Um índice com mais de três colunas é quase certo não ser útil, a menos que a utilização da tabela seja muito peculiar.

## 11.4. Índices únicos

Os índices também podem ser utilizados para impor a unicidade do valor de uma coluna, ou a unicidade dos valores combinados de mais de uma coluna.

```
CREATE UNIQUE INDEX nome ON tabela (coluna [, ...]);
```

Atualmente, somente os índices B-tree poder ser declarados como únicos.

Quando o índice é declarado como único, não pode existir na tabela mais de uma linha com valores indexados iguais. Os valores nulos não são considerados iguais. Um índice único com várias colunas rejeita apenas os casos onde todas as colunas indexadas são iguais em duas linhas.

O PostgreSQL cria, automaticamente, um índice único quando é definida na tabela uma restrição de unicidade ou uma chave primária. O índice abrange as colunas que compõem a chave primária ou as colunas únicas (um índice com várias colunas, se for apropriado), sendo este o mecanismo que impõe a restrição.

**Nota:** A forma preferida para adicionar restrição de unicidade a uma tabela é por meio do comando `ALTER TABLE ... ADD CONSTRAINT`. A utilização de índices para impor restrições de unicidade pode ser considerada um detalhe de implementação que não deve ser acessado diretamente. Entretanto, deve-se ter em mente que não é necessário criar índices em colunas únicas manualmente; caso se faça, simplesmente se duplicará o índice criado automaticamente.

## 11.5. Índices em expressões

Uma coluna do índice não precisa ser apenas uma coluna da tabela subjacente, pode ser uma função ou uma expressão escalar computada a partir de uma ou mais colunas da tabela. Esta funcionalidade é útil para obter acesso rápido às tabelas com base em resultados de cálculos.<sup>3</sup>

Por exemplo, uma forma habitual de fazer comparações não diferenciando letras maiúsculas de minúsculas é utilizar a função `lower`:

```
SELECT * FROM teste1 WHERE lower(col1) = 'valor';
```

```
-- Para não diferenciar maiúsculas e minúsculas, acentuadas ou não (N. do T.)
```

```
SELECT * FROM teste1 WHERE lower(to_ascii(col1)) = 'valor';
```

Esta consulta pode utilizar um índice, caso algum tenha sido definido sobre o resultado da operação `lower(col1)`:

```
CREATE INDEX idx_teste1_lower_col1 ON teste1 (lower(col1));
```

```
-- Para incluir as letras acentuadas (N. do T.)
```

```
CREATE INDEX idx_teste1_lower_ascii_col1 ON teste1 (lower(to_ascii(col1)));
```

Se o índice for declarado como `UNIQUE`, este impede a criação de linhas cujos valores de `col1` diferem apenas em letras maiúsculas e minúsculas, assim como a criação de linhas cujos valores de `col1` são realmente idênticos. Portanto, podem ser utilizados índices em expressões para impor restrições que não podem ser definidas como restrições simples de unicidade.

Como outro exemplo, quando são feitas habitualmente consultas do tipo

```
SELECT * FROM pessoas WHERE (primeiro_nome || ' ' || ultimo_nome) = 'Manoel Silva';
```



então vale a pena criar um índice como:

```
CREATE INDEX idx_pessoas_nome ON pessoas ((primeiro_nome || ' ' || ultimo_nome));
```

A sintaxe do comando `CREATE INDEX` normalmente requer que se escreva parênteses em torno da expressão do índice, conforme mostrado no segundo exemplo. Os parênteses podem ser omitidos quando a expressão for apenas uma chamada de função, como no primeiro exemplo.

É relativamente dispendioso manter expressões de índice, uma vez que a expressão derivada deve ser computada para cada linha inserida, ou sempre que for atualizada. Portanto, devem ser utilizadas somente quando as consultas que usam o índice são muito freqüentes.

## 11.6. Classes de operadores

A definição do índice pode especificar uma *classe de operadores* para cada coluna do índice.

```
CREATE INDEX nome ON tabela (coluna classe_de_operadores [, ...]);
```

A classe de operadores identifica os operadores a serem utilizados pelo índice para esta coluna. Por exemplo, um índice B-tree no tipo `int4` utiliza a classe `int4_ops`; esta classe de operadores inclui funções de comparação para valores do tipo `int4`. Na prática, a classe de operadores padrão para o tipo de dado da coluna é normalmente suficiente. O ponto principal de existir classes de operadores é que, para alguns tipos de dado, pode haver mais de um comportamento do índice que faça sentido. Por exemplo, pode-se desejar ordenar o tipo de dado do número complexo tanto pelo valor absoluto quanto pela parte real, o que pode ser feito definindo duas classes de operadores para o tipo de dado e, depois, selecionando a classe apropriada ao definir o índice.

Existem, também, algumas classes de operadores nativas além das classes padrão:

- As classes de operadores `text_pattern_ops`, `varchar_pattern_ops`, `bpchar_pattern_ops` e `name_pattern_ops` dão suporte a índices B-tree nos tipos `text`, `varchar`, `char` e `name`, respectivamente. A diferença com relação às classes de operadores comuns, é que os valores são comparados estritamente caractere por caractere, em vez de seguir as regras de classificação (*collation*<sup>4</sup>) específicas do idioma. Isto torna estas classes de operadores adequadas para serem usadas em comandos envolvendo expressões de correspondência com padrão (expressões regulares `LIKE` ou `POSIX`) se o servidor não usar o idioma “C” padrão. Como exemplo, uma coluna `varchar` pode ser indexada da seguinte forma:

```
CREATE INDEX idx_teste ON tbl_teste (col varchar_pattern_ops);
```

Se for utilizado o idioma C, em vez disso pode ser criado um índice com a classe de operadores padrão, e ainda assim será útil para comandos com correspondência com padrão. Deve ser observado, também, que deve ser criado um índice com a classe de operadores padrão se for desejado que consultas envolvendo comparações comuns utilizem um índice. Estas consultas não podem utilizar a classe de operadores `xxx_pattern_ops`. Podem ser criados vários índices na mesma coluna com classes de operadores diferentes.

A consulta a seguir mostra todas as classes de operadores definidas:

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name
FROM pg_am am, pg_opclass opc
WHERE opc.opcamid = am.oid
ORDER BY index_method, opclass_name;
```

Podendo ser estendida para mostrar todos os operadores incluídos em cada classe:

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opr.oprname AS opclass_operator
FROM pg_am am, pg_opclass opc, pg_amop amop, pg_operator opr
WHERE opc.opcamid = am.oid AND
      amop.amopclaid = opc.oid AND
      amop.amopopr = opr.oid
ORDER BY index_method, opclass_name, opclass_operator;
```

## 11.7. Índices parciais

O *índice parcial* é um índice construído sobre um subconjunto da tabela; o subconjunto é definido por uma expressão condicional (chamada de *predicado* <sup>5</sup> do índice parcial). O índice contém entradas apenas para as linhas da tabela que satisfazem o predicado. <sup>6</sup>

O principal motivo para criar índices parciais é evitar a indexação de valores freqüentes. Como um comando procurando por um valor freqüente (um que apareça em mais que uma pequena percentagem de linhas da tabela) não utiliza o índice de qualquer forma, não faz sentido manter estas linhas no índice. Isto reduz o tamanho do índice, acelerando as consultas que utilizam este índice. Também acelera muitas operações de atualização da tabela, porque o índice não precisa ser atualizado em todos os casos. O Exemplo 11-1 mostra uma aplicação possível desta idéia.

### Exemplo 11-1. Definir um índice parcial excluindo valores freqüentes

Suponha que os registros de acesso ao servidor web são armazenadas no banco de dados, e que a maioria dos acessos se origina na faixa de endereços de IP da própria organização, mas alguns são de fora (digamos, empregados com acesso discado). Se a procura por endereços de IP for principalmente sobre acesso externo, provavelmente não será necessário indexar a faixa de endereços de IP correspondente à subrede da própria organização.

Assumindo que exista uma tabela como esta:

```
CREATE TABLE tbl_registro_acesso (
    url          varchar,
    ip_cliente   inet,
    ...
);
```

Para criar um índice parcial adequado ao exemplo acima, deve ser utilizado um comando como:

```
CREATE INDEX idx_registro_acesso_ip_cliente ON tbl_registro_acesso (ip_cliente)
WHERE NOT (ip_cliente > inet '192.168.100.0' AND ip_cliente < inet '192.168.100.255');
```

Uma consulta típica que pode utilizar este índice é:

```
SELECT * FROM tbl_registro_acesso WHERE url = '/index.html' AND ip_cliente = inet
'212.78.10.32';
```

Uma consulta típica que não pode utilizar este índice é:

```
SELECT * FROM access_log WHERE client_ip = inet '192.168.100.23';
```

Deve ser observado que este tipo de índice parcial requer que os valores comuns sejam determinados a priori. Se a distribuição dos valores for inerente (devido à natureza da aplicação) e estática (não muda com o tempo) não é difícil, mas se os valores freqüentes se devem meramente à carga de dados coincidentes, pode ser necessário bastante trabalho de manutenção.

Outra possibilidade é excluir do índice os valores para os quais o perfil típico das consultas não tenha interesse, conforme mostrado no Exemplo 11-2. Isto resulta nas mesmas vantagens mostradas acima, mas impede o acesso aos valores “que não interessam” por meio deste índice, mesmo se a varredura do índice for vantajosa neste caso. Obviamente, definir índice parcial para este tipo de cenário requer muito cuidado e experimentação.

### Exemplo 11-2. Definir um índice parcial excluindo valores que não interessam

Se existir uma tabela contendo tanto pedidos faturados quanto não faturados, onde os pedidos não faturados representam uma pequena parte da tabela, mas são os mais acessados, é possível melhorar o desempenho criando um índice somente para os pedidos não faturados. O comando para criar o índice deve ser parecido com este:

```
CREATE INDEX idx_pedidos_ao_faturados ON pedidos (num_pedido)
WHERE faturado is not true;
```

Uma possível consulta utilizando este índice é

```
SELECT * FROM pedidos WHERE faturado is not true AND num_pedido < 10000;
```

Entretanto, o índice também pode ser utilizado em consultas não envolvendo num\_pedido como, por exemplo,

```
SELECT * FROM pedidos WHERE faturado is not true AND valor > 5000.00;
```

Embora não seja tão eficiente quanto seria um índice parcial na coluna `valor`, porque o sistema precisa percorrer o índice por inteiro, mesmo assim, havendo poucos pedidos não faturados, a utilização do índice parcial para localizar apenas os pedidos não faturados pode ser vantajosa.

Deve ser observado que a consulta abaixo não pode utilizar este índice:

```
SELECT * FROM pedidos WHERE num_pedido = 3501;
```

O pedido número 3501 pode estar entre os pedidos faturados e os não faturados.

O Exemplo 11-2 também ilustra que a coluna indexada e a coluna utilizada no predicado não precisam corresponder. O PostgreSQL suporta índices parciais com predicados arbitrários, desde que somente estejam envolvidas colunas da tabela indexada. Entretanto, deve-se ter em mente que o predicado deve corresponder às condições utilizadas nos comandos que supostamente vão se beneficiar do índice. Para ser preciso, o índice parcial somente pode ser utilizado em um comando se o sistema puder reconhecer que a condição `WHERE` do comando implica matematicamente no predicado do índice. O PostgreSQL não possui um provador de teoremas sofisticado que possa reconhecer expressões equivalentes matematicamente escritas de forma diferente (Não seria apenas extremamente difícil criar este provador de teoremas geral, como este provavelmente também seria muito lento para poder ser usado na prática). O sistema pode reconhecer implicações de desigualdades simples como, por exemplo, “ $x < 1$ ” implica “ $x < 2$ ”; senão, a condição do predicado deve corresponder exatamente a uma parte da condição `WHERE` da consulta, ou o índice não será reconhecido como utilizável.

Um terceiro uso possível para índices parciais não requer que o índice seja utilizado em nenhum comando. A idéia é criar um índice único sobre um subconjunto da tabela, como no Exemplo 11-3, impondo a unicidade das linhas que satisfazem o predicado do índice, sem restringir as que não fazem parte.

### Exemplo 11-3. Definir um índice único parcial

Suponha que exista uma tabela contendo perguntas e respostas. Deseja-se garantir que exista apenas uma resposta “correta” para uma dada pergunta, mas que possa haver qualquer número de respostas “incorretas”. Abaixo está mostrada a forma de fazer:

```
CREATE TABLE tbl_teste
(
    pergunta    text,
    resposta    text,
    correto     bool
    ...
);

CREATE UNIQUE INDEX unq_resposta_correta ON tbl_teste (pergunta, correto)
WHERE correto;
```

Esta forma é particularmente eficiente quando existem poucas respostas corretas, e muitas incorretas.

Finalizando, também pode ser utilizado um índice parcial para mudar a escolha do plano de comando feito pelo sistema. Pode ocorrer que conjuntos de dados com distribuições peculiares façam o sistema utilizar um índice quando na realidade não deveria. Neste caso, o índice pode ser definido de modo que não esteja disponível para o comando com problema. Normalmente, o PostgreSQL realiza escolhas razoáveis com relação à utilização dos índices (por exemplo, evita-os ao buscar valores com muitas ocorrências, desta maneira o primeiro exemplo realmente economiza apenas o tamanho do índice, mas não é necessário para evitar a utilização do índice), e a escolha de um plano grosseiramente incorreto é motivo para um relatório de erro.

Deve-se ter em mente que a criação de um índice parcial indica que você sabe pelo menos tanto quanto o planejador de comandos sabe. Em particular, você sabe quando um índice poderá ser vantajoso. A formação deste conhecimento requer experiência e compreensão sobre como os índices funcionam no PostgreSQL. Na maioria dos casos, a vantagem de um índice parcial sobre um índice regular não é muita.

Podem ser obtidas informações adicionais sobre índices parciais em *The case for partial indexes*, *Partial indexing in POSTGRES: research project* e *Generalized Partial Indexes*.

## 11.8. Examinar a utilização do índice

Embora no PostgreSQL os índices não necessitem de manutenção e ajuste, ainda assim é importante verificar quais índices são utilizados realmente pelos comandos executados no ambiente de produção. O exame da utilização de um índice por um determinado comando é feito por meio do comando `EXPLAIN`; sua aplicação para esta finalidade está ilustrada na Seção

13.1. Também é possível coletar estatísticas gerais sobre a utilização dos índices por um servidor em operação da maneira descrita na Seção 23.2.

É difícil formular um procedimento genérico para determinar quais índices devem ser definidos. Existem vários casos típicos que foram mostrados nos exemplos das seções anteriores. Muita verificação experimental é necessária na maioria dos casos. O restante desta seção dá algumas dicas.

- O comando *ANALYZE* sempre deve ser executado primeiro. Este comando coleta estatísticas sobre a distribuição dos valores na tabela. Esta informação é necessária para estimar o número de linhas retornadas pela consulta, que é uma necessidade do planejador para atribuir custos dentro da realidade para cada plano de comando possível. Na ausência de estatísticas reais, são assumidos alguns valores padrão, quase sempre imprecisos. O exame da utilização do índice pelo aplicativo sem a execução prévia do comando *ANALYZE* é, portanto, uma causa perdida.
- Devem ser usados dados reais para a verificação experimental. O uso de dados de teste para definir índices diz quais índices são necessários para os dados de teste, e nada além disso.

É especialmente fatal utilizar conjuntos de dados de teste muito pequenos. Enquanto selecionar 1.000 de cada 100.000 linhas pode ser um candidato para um índice, selecionar 1 de cada 100 linhas dificilmente será, porque as 100 linhas provavelmente cabem dentro de uma única página do disco, e não existe nenhum plano melhor que uma busca seqüencial em uma página do disco.

Também deve ser tomado cuidado ao produzir os dados de teste, geralmente não disponíveis quando o aplicativo ainda não se encontra em produção. Valores muito semelhantes, completamente aleatórios, ou inseridos ordenadamente, distorcem as estatísticas em relação à distribuição que os dados reais devem ter.

- Quando os índices não são usados, pode ser útil como teste forçar sua utilização. Existem parâmetros em tempo de execução que podem desabilitar vários tipos de planos (descritos no Seção 16.4). Por exemplo, desabilitar varreduras seqüenciais (*enable\_seqscan*) e junções de laço-aninhado (*enable\_nestloop*), que são os planos mais básicos, forcem o sistema a utilizar um plano diferente. Se o sistema ainda assim escolher a varredura seqüencial ou a junção de laço-aninhado então existe, provavelmente, algum problema mais fundamental devido ao qual o índice não está sendo utilizado como, por exemplo, a condição da consulta não corresponde ao índice (Qual tipo de consulta pode utilizar qual tipo de índice é explicado nas seções anteriores).
- Se forçar a utilização do índice não faz o índice ser usado, então existem duas possibilidades: ou o sistema está correto e realmente a utilização do índice não é apropriada, ou a estimativa de custo dos planos de comando não estão refletindo a realidade. Portanto, deve ser medido o tempo da consulta com e sem índices. O comando *EXPLAIN ANALYZE* pode ser útil neste caso.
- Se for descoberto que as estimativas de custo estão erradas existem, novamente, duas possibilidades. O custo total é calculado a partir do custo por linha de cada nó do plano vezes a seletividade estimada do nó do plano. Os custos dos nós do plano podem ser ajustados usando parâmetros em tempo de execução (descritos no Seção 16.4). A estimativa imprecisa da seletividade é devida a estatísticas insuficientes. É possível melhorar esta situação ajustando os parâmetros de captura de estatísticas (consulte o comando *ALTER TABLE*).

Se não for obtido sucesso no ajuste dos custos para ficarem mais apropriados, então pode ser necessário o recurso de forçar a utilização do índice explicitamente. Pode-se, também, desejar fazer contato com os desenvolvedores do PostgreSQL para examinar este problema.

## Notas

1. *hashing* — valor de identificação produzido através da execução de uma operação numérica, denominada função de *hashing*, em um item de dado. O valor identifica de forma exclusiva o item de dado, mas exige um espaço de armazenamento bem menor. Por isso, o computador pode localizar mais rapidamente os valores de *hashing* que os itens de dado, que são mais extensos. Uma tabela de *hashing* associa cada valor a um item de dado exclusivo. Webster's New World Dicionário de Informática, Brian Pfaffenberger, Editora Campus, 1999. (N. do T.)
2. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
3. O sistema gerenciador de banco de dados Oracle 10g também permite usar função e expressão escalar na coluna do índice, mas o SQL Server 2000 e o DB2 8.1 não permitem. Comparison of relational database management systems (<http://www.answers.com/topic/comparison-of-relational-database-management-systems>) (N. do T.)

4. *collation*; *collating sequence* — Um método para comparar duas cadeias de caracteres comparáveis. Todo conjunto de caracteres possui seu *collation* padrão. (Second Informal Review Draft) ISO/IEC 9075:1992, Database Language SQL- July 30, 1992. (N. do T.)
5. *predicado* — especifica uma condição que pode ser avaliada para obter um resultado booleano. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
6. Os sistemas gerenciadores de banco de dados SQL Server 2000, Oracle 10g e DB2 8.1 não possuem suporte a índices parciais. Comparison of relational database management systems (<http://www.answers.com/topic/comparison-of-relational-database-management-systems>) (N. do T.)

# Capítulo 12. Controle de simultaneidade

Este capítulo descreve o comportamento do sistema gerenciador de banco de dados PostgreSQL quando duas ou mais sessões tentam acessar os mesmos dados ao mesmo tempo. O objetivo nesta situação é permitir acesso eficiente para todas as sessões mantendo, ao mesmo tempo, uma rigorosa integridade dos dados. Todos os desenvolvedores de aplicativos de banco de dados devem estar familiarizados com os tópicos cobertos por este capítulo.

## 12.1. Introdução

Diferentemente dos sistemas gerenciadores de banco de dados tradicionais, que usam bloqueios para controlar a simultaneidade, o PostgreSQL mantém a consistência dos dados utilizando o modelo multiversão (Multiversion Concurrency Control, MVCC). Isto significa que, ao consultar o banco de dados cada transação enxerga um instantâneo (snapshot) dos dados (uma *versão do banco de dados*) como estes eram há um tempo atrás, sem levar em consideração o estado corrente dos dados subjacentes. Este modelo protege a transação contra enxergar dados inconsistentes, o que poderia ser causado por atualizações feitas por transações simultâneas nas mesmas linhas de dados, fornecendo um *isolamento da transação* para cada sessão do banco de dados.

A principal vantagem de utilizar o modelo de controle de simultaneidade MVCC em vez de bloqueios é que, no MVCC os bloqueios obtidos para consultar dados (leitura) não conflitam com os bloqueios obtidos para escrever dados e, portanto, a leitura nunca bloqueia a escrita, e a escrita nunca bloqueia a leitura.

Também estão disponíveis no PostgreSQL as funcionalidades de bloqueio no nível de tabela e de linha, para aplicativos que não podem se adaptar facilmente ao comportamento MVCC. Entretanto, a utilização apropriada do MVCC geralmente produz um desempenho melhor que os bloqueios.

## 12.2. Isolamento da transação

O padrão SQL define quatro níveis de isolamento de transação em termos de três fenômenos que devem ser evitados entre transações simultâneas. Os fenômenos não desejados são:

`dirty read` (leitura suja)

A transação lê dados escritos por uma transação simultânea não efetivada (`uncommitted`).<sup>1</sup>

`nonrepeatable read` (leitura que não pode ser repetida)

A transação lê novamente dados lidos anteriormente, e descobre que os dados foram alterados por outra transação (que os efetivou após ter sido feita a leitura anterior).<sup>2</sup>

`phantom read` (leitura fantasma)

A transação executa uma segunda vez uma consulta que retorna um conjunto de linhas que satisfazem uma determinada condição de procura, e descobre que o conjunto de linhas que satisfazem a condição é diferente por causa de uma outra transação efetivada recentemente.<sup>3</sup>

Os quatro níveis de isolamento de transação, e seus comportamentos correspondentes, estão descritos na Tabela 12-1.

**Tabela 12-1. Níveis de isolamento da transação no SQL**

Nível de isolamento	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possível	Possível	Possível
Read committed	Impossível	Possível	Possível
Repeatable read	Impossível	Impossível	Possível
Serializable	Impossível	Impossível	Impossível

No PostgreSQL pode ser requisitado qualquer um dos quatro níveis de isolamento padrão. Porém, internamente só existem dois níveis de isolamento distintos, correspondendo aos níveis de isolamento `Read Committed` e `Serializable`. Quando é selecionado o nível de isolamento `Read Committed` realmente obtém-se `Read Committed`, mas quando é selecionado `Repeatable Read` na realidade é obtido `Serializable`. Portanto, o nível de isolamento real pode ser mais estrito do que o selecionado. Isto é permitido pelo padrão SQL: os quatro níveis de isolamento somente definem quais fenômenos não podem acontecer, não definem quais fenômenos devem acontecer. O motivo pelo qual o PostgreSQL só disponibiliza dois níveis de isolamento, é porque esta é a única forma de mapear os níveis de isolamento padrão na arquitetura de controle de simultaneidade multiversão que faz sentido. O comportamento dos níveis de isolamento disponíveis estão detalhados nas próximas subseções.

É utilizado o comando `SET TRANSACTION` para definir o nível de isolamento da transação.

### 12.2.1. Nível de isolamento `Read Committed`

O `Read Committed` (lê efetivado) é o nível de isolamento padrão do PostgreSQL. Quando uma transação processa sob este nível de isolamento, o comando `SELECT` enxerga apenas os dados efetivados antes da consulta começar; nunca enxerga dados não efetivados, ou as alterações efetivadas pelas transações simultâneas durante a execução da consulta (Entretanto, o `SELECT` enxerga os efeitos das atualizações anteriores executadas dentro da sua própria transação, mesmo que ainda não tenham sido efetivadas). Na verdade, o comando `SELECT` enxerga um instantâneo do banco de dados, como este era no instante em que a consulta começou a executar. Deve ser observado que dois comandos `SELECT` sucessivos podem enxergar dados diferentes, mesmo estando dentro da mesma transação, se outras transações efetivarem alterações durante a execução do primeiro comando `SELECT`.

Os comandos `UPDATE`, `DELETE` e `SELECT FOR UPDATE` se comportam do mesmo modo que o `SELECT` para encontrar as linhas de destino: somente encontram linhas de destino efetivadas até o momento do início do comando. Entretanto, no momento em que foi encontrada alguma linha de destino pode ter sido atualizada (ou excluída ou marcada para atualização) por outra transação simultânea. Neste caso, a transação que pretende atualizar fica aguardando a transação de atualização que começou primeiro efetivar ou desfazer (se ainda estiver executando). Se a transação de atualização que começou primeiro desfizer as atualizações, então seus efeitos são negados e a segunda transação de atualização pode prosseguir com a atualização da linha original encontrada. Se a transação de atualização que começou primeiro efetivar as atualizações, a segunda transação de atualização ignora a linha caso tenha sido excluída pela primeira transação de atualização, senão tenta aplicar sua operação na versão atualizada da linha. A condição de procura do comando (a cláusula `WHERE`) é avaliada novamente para verificar se a versão atualizada da linha ainda corresponde à condição de procura. Se corresponder, a segunda transação de atualização prossegue sua operação começando a partir da versão atualizada da linha.

Devido à regra acima, é possível um comando de atualização enxergar um instantâneo inconsistente: pode enxergar os efeitos dos comandos simultâneos de atualização que afetam as mesmas linhas que está tentando atualizar, mas não enxerga os efeitos destes comandos de atualização nas outras linhas do banco de dados. Este comportamento torna o `Read Committed` inadequado para os comandos envolvendo condições de procura complexas. Entretanto, é apropriado para casos mais simples. Por exemplo, considere a atualização do saldo bancário pela transação mostrada abaixo:

```
BEGIN;
UPDATE conta SET saldo = saldo + 100.00 WHERE num_conta = 12345;
UPDATE conta SET saldo = saldo - 100.00 WHERE num_conta = 7534;
COMMIT;
```

Se duas transações deste tipo tentam mudar simultaneamente o saldo da conta 12345, é claro que se deseja que a segunda transação comece a partir da versão atualizada da linha da conta. Como cada comando afeta apenas uma linha predeterminada, permitir enxergar a versão atualizada da linha não cria nenhum problema de inconsistência.

Como no modo `Read Committed` cada novo comando começa com um novo instantâneo incluindo todas as transações efetivadas até este instante, de qualquer modo os próximos comandos na mesma transação vão enxergar os efeitos das transações simultâneas efetivadas. O ponto em questão é se, dentro de um *único* comando, é enxergada uma visão totalmente consistente do banco de dados.

O isolamento parcial da transação fornecido pelo modo `Read Committed` é adequado para muitos aplicativos, e este modo é rápido e fácil de ser utilizado. Entretanto, para aplicativos que efetuam consultas e atualizações complexas, pode ser necessário garantir uma visão do banco de dados com consistência mais rigorosa que a fornecida pelo modo `Read Committed`.

### 12.2.2. Nível de isolamento serializável

O nível *Serializable* fornece o isolamento de transação mais rigoroso. Este nível emula a execução serial das transações, como se todas as transações fossem executadas uma após a outra, em série, em vez de simultaneamente. Entretanto, os aplicativos que utilizam este nível de isolamento devem estar preparados para tentar executar novamente as transações, devido a falhas de serialização.

Quando uma transação está no nível serializável, o comando `SELECT` enxerga apenas os dados efetivados antes da transação começar; nunca enxerga dados não efetivados ou alterações efetivadas durante a execução da transação por transações simultâneas (Entretanto, o comando `SELECT` enxerga os efeitos das atualizações anteriores executadas dentro da sua própria transação, mesmo que ainda não tenham sido efetivadas). É diferente do `Read Committed`, porque o comando `SELECT` enxerga um instantâneo do momento de início da transação, e não do momento de início do comando corrente dentro da transação. Portanto, comandos `SELECT` sucessivos dentro de uma mesma transação sempre enxergam os mesmos dados.

Os comandos `UPDATE`, `DELETE` e `SELECT FOR UPDATE` se comportam do mesmo modo que o comando `SELECT` para encontrar as linhas de destino: somente encontram linhas de destino efetivadas até o momento do início da transação. Entretanto, alguma linha de destino pode ter sido atualizada (ou excluída ou marcada para atualização) por outra transação simultânea no momento em que foi encontrada. Neste caso, a transação serializável aguarda a transação de atualização que começou primeiro efetivar ou desfazer as alterações (se ainda estiver executando). Se a transação que começou primeiro desfizer as alterações, então seus efeitos são negados e a transação serializável pode prosseguir com a atualização da linha original encontrada. Porém, se a transação que começou primeiro efetivar (e realmente atualizar ou excluir a linha, e não apenas selecionar para atualização), então a transação serializável é desfeita com a mensagem

ERRO: não foi possível serializar o acesso devido a atualização simultânea

porque uma transação serializável não pode alterar linhas alteradas por outra transação após a transação serializável ter começado.

Quando o aplicativo receber esta mensagem de erro deverá interromper a transação corrente, e tentar executar novamente toda a transação a partir do início. Da segunda vez em diante, a transação passa a enxergar a alteração efetivada anteriormente como parte da sua visão inicial do banco de dados e, portanto, não existirá conflito lógico em usar a nova versão da linha como ponto de partida para atualização na nova transação.

Deve ser observado que somente as transações que fazem atualizações podem precisar de novas tentativas; as transações somente para leitura nunca estão sujeitas a conflito de serialização.

O modo serializável fornece uma garantia rigorosa que cada transação enxerga apenas visões totalmente consistentes do banco de dados. Entretanto, o aplicativo deve estar preparado para executar novamente a transação quando atualizações simultâneas tornarem impossível sustentar a ilusão de uma execução serial. Como o custo de refazer transações complexas pode ser significativo, este modo é recomendado somente quando as transações efetuando atualizações contêm lógica suficientemente complexa a ponto de produzir respostas erradas no modo `Read Committed`. Habitualmente, o modo serializável é necessário quando a transação executa vários comandos sucessivos que necessitam enxergar visões idênticas do banco de dados.

#### 12.2.2.1. Isolamento serializável versus verdadeira serialidade

O significado intuitivo (e a definição matemática) de execução “serializável” é que quaisquer duas transações simultâneas efetivadas com sucesso parecem ter sido executadas de forma rigorosamente serial, uma após a outra — embora qual das duas parece ter ocorrido primeiro não pode ser previsto antecipadamente. É importante ter em mente que proibir os comportamentos indesejáveis listados na Tabela 12-1 não é suficiente para garantir a verdadeira serialidade e, de fato, o modo serializável do PostgreSQL *não garante a execução serializável neste sentido*. Como exemplo será considerada a tabela `minha_tabela` contendo inicialmente

classe	valor
1	10
1	20
2	100
2	200



Suponha que a transação serializável A calcula

```
SELECT SUM(valor) FROM minha_tabela WHERE classe = 1;
```

e insira o resultado (30) como `valor` em uma nova linha com `classe = 2`. Simultaneamente a transação serializável B calcula

```
SELECT SUM(valor) FROM minha_tabela WHERE classe = 2;
```

e obtém o resultado 300, que é inserido em uma nova linha com `classe = 1`. Em seguida as duas transações efetivam. Nenhum dos comportamentos não desejados ocorreu, ainda assim foi obtido um resultado que não poderia ter ocorrido serialmente em qualquer ordem. Se A tivesse executado antes de B, então B teria calculado a soma como 330, e não 300, e de maneira semelhante a outra ordem teria produzido uma soma diferente na transação A.

Para garantir serialidade matemática verdadeira, é necessário que o sistema de banco de dados imponha o *bloqueio de predicado*, significando que a transação não pode inserir ou alterar uma linha que corresponde à condição `WHERE` de um comando de outra transação simultânea. Por exemplo, uma vez que a transação A tenha executado o comando `SELECT ... WHERE class = 1`, o sistema de bloqueio de predicado proibiria a transação B inserir qualquer linha com classe igual a 1 até que A fosse efetivada.<sup>4</sup> Um sistema de bloqueio deste tipo é de implementação complexa e de execução extremamente dispendiosa, uma vez que todas as sessões devem estar cientes dos detalhes de todos os comandos executados por todas as transações simultâneas. E este grande gasto em sua maior parte seria desperdiçado, porque na prática a maioria dos aplicativos não fazem coisas do tipo que podem ocasionar problemas (Certamente o exemplo acima é bastante irreal, dificilmente representando um programa de verdade). Portanto, o PostgreSQL não implementa o bloqueio de predicado, e tanto quanto saibamos nenhum outro SGBD de produção o faz.

Nos casos em que a possibilidade de execução não serial representa um perigo real, os problemas podem ser evitados através da utilização apropriada de bloqueios explícitos. São mostrados mais detalhes nas próximas seções.

## 12.3. Bloqueio explícito

O PostgreSQL fornece vários modos de bloqueio para controlar o acesso simultâneo aos dados nas tabelas. Estes modos podem ser utilizados para controlar o bloqueio pela transação, nas situações onde o MVCC não produz o comportamento adequado. Também, a maioria dos comandos do PostgreSQL obtém, automaticamente, bloqueios nos modos apropriados para garantir que as tabelas referenciadas não serão excluídas ou alteradas de forma incompatível enquanto o comando estiver executando (Por exemplo, o comando `ALTER TABLE` não pode executar simultaneamente com outras operações na mesma tabela).

Para examinar a lista de bloqueios ativos no servidor de banco de dados em um determinado instante, deve ser utilizada a visão do sistema `pg_locks` (Seção 41.33). Para obter informações adicionais sobre a monitoração do status do subsistema de gerência de bloqueios consulte o Capítulo 23.

### 12.3.1. Bloqueios no nível de tabela

A lista abaixo mostra os modos de bloqueio disponíveis, e os contextos onde estes modos são utilizados automaticamente pelo PostgreSQL. Também pode ser obtido explicitamente qualquer um destes níveis de bloqueio através do comando `LOCK`. Lembre-se que todos estes modos de bloqueio são no nível de tabela, mesmo que o nome contenha a palavra “row” (linha). Os nomes dos modos de bloqueio são históricos. De alguma forma os nomes refletem a utilização típica de cada modo de bloqueio — mas as semânticas são todas a mesma. A única diferença real entre um modo de bloqueio e outro é o conjunto de modos de bloqueio que cada um conflita. Duas transações não podem obter bloqueios com modos conflitantes na mesma tabela ao mesmo tempo (Entretanto, uma transação nunca conflita consigo mesma. Por exemplo, pode obter o bloqueio `ACCESS EXCLUSIVE` e posteriormente obter o bloqueio `ACCESS SHARE` na mesma tabela). Podem ser obtidos simultaneamente modos de bloqueio não conflitantes por muitas transações. Em particular, deve ser observado que alguns modos de bloqueio são autoconflitantes (por exemplo, o modo de bloqueio `ACCESS EXCLUSIVE` não pode ser obtido por mais de uma transação ao mesmo tempo), enquanto outros não são autoconflitantes (por exemplo, o modo de bloqueio `ACCESS SHARE` pode ser obtido por várias transações ao mesmo tempo). Uma vez obtido, o modo de bloqueio permanece até o fim da transação.

## Modos de bloqueio no nível de tabela

### ACCESS SHARE

Conflita apenas com o modo de bloqueio ACCESS EXCLUSIVE.

O comando SELECT e o comando ANALYZE obtêm um bloqueio neste modo nas tabelas referenciadas. Em geral, qualquer comando que apenas lê a tabela sem modificá-la obtém este modo de bloqueio.

### ROW SHARE

Conflita com os modos de bloqueio EXCLUSIVE e ACCESS EXCLUSIVE.

O comando SELECT FOR UPDATE obtém o bloqueio neste modo na(s) tabela(s) de destino (além do bloqueio no modo ACCESS SHARE para as demais tabelas referenciadas mas não selecionadas FOR UPDATE).

### ROW EXCLUSIVE

Conflita com os modos de bloqueio SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE.

Os comandos UPDATE, DELETE e INSERT obtêm este modo de bloqueio na tabela de destino (além do modo de bloqueio ACCESS SHARE nas outras tabelas referenciadas). Em geral, este modo de bloqueio é obtido por todos os comandos que alteram os dados da tabela.

### SHARE UPDATE EXCLUSIVE

Conflita com os modos de bloqueio SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE. Este modo protege a tabela contra alterações simultâneas no esquema e a execução do comando VACUUM.

Obtida pelo comando VACUUM (sem a opção FULL).

### SHARE

Conflita com os modos de bloqueio ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE. Este modo protege a tabela contra alterações simultâneas nos dados.

Obtido pelo comando CREATE INDEX.

### SHARE ROW EXCLUSIVE

Conflita com os modos de bloqueio ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE.

Este modo de bloqueio não é obtido automaticamente por nenhum comando do PostgreSQL.

### EXCLUSIVE

Conflita com os modos de bloqueio ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE. Este modo permite apenas bloqueios ACCESS SHARE simultâneos, ou seja, somente leituras da tabela podem prosseguir em paralelo com uma transação que obteve este modo de bloqueio.

Este modo de bloqueio não é obtido automaticamente por nenhum comando do PostgreSQL. Entretanto, é obtido em alguns catálogos do sistema em algumas operações.

### ACCESS EXCLUSIVE

Conflita com todos os modos de bloqueio (ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE). Este modo garante que a transação que o obteve é a única acessando a tabela de qualquer forma.

Obtido pelos comandos ALTER TABLE, DROP TABLE, REINDEX, CLUSTER e VACUUM FULL. Este é, também, o modo de bloqueio padrão para o comando LOCK TABLE sem a especificação explícita do modo.

**Dica:** Somente o bloqueio ACCESS EXCLUSIVE bloqueia o comando SELECT (sem a cláusula FOR UPDATE).

## 12.3.2. Bloqueios no nível de linha

Além dos bloqueios no nível de tabela, existem os bloqueios no nível de linha. O bloqueio no nível de linha é obtido automaticamente para uma linha específica quando a linha é atualizada (ou excluída ou marcada para atualização). O

bloqueio é mantido até a transação efetivar ou desfazer as alterações. Os bloqueios no nível de linha não afetam a consulta aos dados; bloqueiam apenas *escritas na mesma linha*. Para obter um bloqueio no nível de linha sem na verdade modificá-la, a linha deve ser selecionada por meio do comando `SELECT FOR UPDATE`. Deve ser observado que após ser obtido um bloqueio no nível de linha, a transação pode atualizar esta linha várias vezes sem medo de conflito.

O PostgreSQL não guarda em memória qualquer informação sobre as linhas alteradas, portanto não existe limite de número de linhas bloqueadas de cada vez. Entretanto, o bloqueio de uma linha pode causar escrita no disco; por exemplo, o comando `SELECT FOR UPDATE` altera as linhas selecionadas para marcá-las, ocasionando escrita em disco.

Além dos bloqueios de tabela e de linha, também são utilizados bloqueios compartilhados e exclusivos no nível de página, para controlar o acesso de leitura e escrita nas páginas da tabela no `shared buffer pool`. Estes bloqueios são liberados imediatamente após a linha ser lida ou atualizada. Normalmente os desenvolvedores de aplicativos não precisam se preocupar com bloqueios no nível de página, sendo mencionados somente para o assunto ficar completo.

### 12.3.3. Impasses

A utilização de bloqueios explícitos pode aumentar a probabilidade de acontecerem *impasses* (*deadlocks*), onde duas (ou mais) transações mantêm bloqueios que a outra deseja. Por exemplo, se a transação 1 obtiver um bloqueio exclusivo na tabela A e, depois, tentar obter um bloqueio exclusivo na tabela B, enquanto a transação 2 já possui um bloqueio exclusivo na tabela B, e agora deseja obter um bloqueio exclusivo na tabela A, então nenhuma das duas transações pode prosseguir. O PostgreSQL detecta automaticamente as situações de impasse, resolvendo-as interrompendo uma das transações envolvidas, permitindo que a(s) outra(s) prossiga(m) (Exatamente qual transação será interrompida é difícil prever, não se devendo confiar nesta previsão).

Deve ser observado que os impasses também podem ocorrer como resultado de um bloqueio no nível de linha (e, por isso, podem ocorrer mesmo que não se use bloqueios explícitos). Considere o caso onde existam duas transações simultâneas alterando uma tabela. A primeira transação executa:

```
UPDATE conta SET saldo = saldo + 100.00 WHERE num_conta = 11111;
```

Este comando obtém um bloqueio no nível de linha na linha com o número da conta especificado. Depois a segunda transação executa:

```
UPDATE conta SET saldo = saldo + 100.00 WHERE num_conta = 22222;
UPDATE conta SET saldo = saldo - 100.00 WHERE num_conta = 11111;
```

O primeiro comando `UPDATE` é bem-sucedido ao obter o bloqueio no nível de linha na linha especificada e, portanto, prossegue atualizando a linha. Entretanto, o segundo comando `UPDATE` descobre que a linha a ser atualizada está bloqueada e, portanto, fica aguardando a transação que obteve o bloqueio terminar. A transação 2 agora está aguardando a transação 1 completar para poder continuar. Agora, a transação 1 executa:

```
UPDATE conta SET saldo = saldo - 100.00 WHERE num_conta = 22222;
```

A transação 1 tenta obter um bloqueio no nível de linha na linha especificada, mas não pode: a transação 2 já possui este bloqueio. Portanto, fica aguardando a transação 2 terminar. Assim sendo, a transação 1 está bloqueada pela transação 2, e a transação 2 está bloqueada pela transação 1: uma condição de impasse. O PostgreSQL detecta esta situação e interrompe uma das transações.

Geralmente a melhor defesa contra os impasses é evitá-los, tendo certeza que todos os aplicativos que utilizam o banco de dados obtêm bloqueios em vários objetos em uma ordem consistente. No exemplo anterior, se as duas transações tivessem atualizado as linhas na mesma ordem, não teria ocorrido nenhum impasse. Deve-se garantir, também, que o primeiro bloqueio obtido em um objeto em uma transação seja aquele com o modo mais alto que será necessário para este objeto. Se for impraticável verificar esta situação antecipadamente, então os impasses podem ser tratados em tempo de execução tentando executar novamente as transações interrompidas pelos impasses.

Enquanto a situação de impasse não for detectada, uma transação em busca de um bloqueio no nível de tabela ou no nível de linha ficará aguardando indefinidamente pela liberação dos bloqueios conflitantes. Isso significa que é uma péssima idéia os aplicativos manterem transações abertas por longos períodos de tempo (por exemplo, aguardando a entrada de dados pelo usuário).

## 12.4. Verificação da consistência dos dados no nível do aplicativo

Como no PostgreSQL a leitura não bloqueia os dados, independentemente do nível de isolamento da transação, os dados lidos por uma transação podem ser sobrescritos por outra transação simultânea. Em outras palavras, se uma linha foi retornada pelo comando `SELECT`, não significa que esta linha ainda era a linha corrente no instante em que foi retornada (ou seja, algum tempo depois do comando corrente ter começado). A linha pode ter sido alterada ou excluída por uma transação já efetivada, que efetivou após esta transação ter começado. Mesmo que a linha ainda seja válida “agora”, esta linha pode ser mudada ou excluída antes da transação corrente efetivar ou desfazer suas alterações.

Outra maneira de pensar sobre isto é que cada transação enxerga um instantâneo do conteúdo do banco de dados, e as transações executando simultaneamente podem, perfeitamente bem, enxergar instantâneos diferentes. Portanto, o próprio conceito de “agora” de alguma forma é mal definido. Normalmente isto não é um grande problema quando os aplicativos cliente estão isolados um do outro, mas se os clientes puderem se comunicar por meio de canais externos ao banco de dados, então podem acontecer sérias confusões.

Para garantir a validade corrente de uma linha e protegê-la contra atualizações simultâneas, deve ser utilizado o comando `SELECT FOR UPDATE` ou uma declaração `LOCK TABLE` apropriada (o comando `SELECT FOR UPDATE` bloqueia apenas as linhas retornadas contra atualizações simultâneas, enquanto o `LOCK TABLE` bloqueia toda a tabela). Isto deve ser levado em consideração ao portar aplicativos de outros ambientes para o PostgreSQL (Antes da versão 6.5 o PostgreSQL utilizava bloqueios de leitura e, portanto, as considerações acima também se aplicam quando é feita a atualização de uma versão do PostgreSQL anterior a 6.5).

As verificações de validade globais requerem considerações adicionais sob o MVCC. Por exemplo, um aplicativo bancário pode desejar verificar se a soma de todos os créditos em uma tabela é igual a soma de todos os débitos em outra tabela, num momento em que as duas tabelas estão sendo ativamente atualizadas. Comparar os resultados de dois comandos `SELECT SUM(...)` sucessivos não funciona confiavelmente no modo `Read Committed`, porque o segundo comando, provavelmente, vai incluir resultados de transações não contabilizadas pelo primeiro comando. Realizar as duas somas em uma mesma transação serializável fornece uma imagem precisa dos efeitos das transações efetivadas antes do início da transação serializável — mas pode ser legitimamente questionado se a resposta ainda era relevante na hora que foi entregue. Se a própria transação serializável introduziu algumas alterações antes de tentar efetuar a verificação de consistência, o valor prático da verificação se torna ainda mais discutível, porque agora são incluídas algumas, mas não todas as alterações ocorridas após o início da transação. Em casos como este, uma pessoa cuidadosa pode desejar bloquear todas as tabelas necessárias para a verificação, para obter uma imagem da situação atual acima de qualquer suspeita. Um bloqueio no modo `SHARE` (ou superior) garante não haver alterações não efetivadas na tabela bloqueada, fora as alterações efetuadas pela própria transação corrente.

Deve ser observado, também, que quando se depende de bloqueios explícitos para evitar alterações simultâneas deve ser utilizado o modo `Read Committed` ou, no modo serializável, tomar o cuidado de obter os bloqueios antes de executar os comandos. Um bloqueio obtido por uma transação serializável garante que nenhuma outra transação alterando a tabela está executando, mas se o instantâneo enxergado pela transação for anterior à obtenção do bloqueio, pode ser que seja anterior a algumas alterações na tabela que agora estão efetivadas. O instantâneo de uma transação serializável é, na verdade, tirado no início da sua primeira consulta ou comando de alteração de dados (`SELECT`, `INSERT`, `UPDATE` ou `DELETE`) sendo, portanto, possível obter bloqueios explicitamente antes do instantâneo ser tirado.

## 12.5. Bloqueio e índices

Embora o PostgreSQL disponibilize acesso de leitura e escrita não bloqueante aos dados das tabelas, o acesso de leitura e escrita não bloqueante não é fornecido, atualmente, por todos os métodos de acesso de índice implementados pelo PostgreSQL. Os vários tipos de índice são tratados como mostrado a seguir:

### Índices B-tree

São utilizados bloqueios no nível de página, compartilhados ou exclusivos, de curta duração, para acesso de leitura/escrita. Os bloqueios são liberados imediatamente após cada linha do índice ser lida ou inserida. Os índices B-tree fornecem a mais alta simultaneidade sem condições de impasse.

### Índices GiST e R-tree

São utilizados bloqueios no nível de índice, compartilhados ou exclusivos, para acesso de leitura/escrita. Os bloqueios são liberados após o comando terminar.

## Índices Hash

São utilizados bloqueios no nível de receptáculo de hash (`hash-bucket-level`), compartilhados ou exclusivos, para acesso de leitura/escrita. Os bloqueios são liberados após todo o receptáculo ser processado. Os bloqueios no nível de receptáculo fornecem uma simultaneidade melhor que os bloqueios no nível de índice, mas podem ocorrer impasses uma vez que os bloqueios são mantidos por mais tempo que uma operação de índice.

Em resumo, o índice `B-tree` oferece o melhor desempenho para aplicações simultâneas; uma vez que também possui mais funcionalidades do que o índice `hash`, é o tipo de índice recomendado para aplicações simultâneas que necessitam indexar dados escalares. Para tratar dados não escalares, os índices `B-tree` obviamente não podem ser utilizados; nesta situação, os desenvolvedores de aplicativos devem estar cientes do desempenho relativamente pobre dos índices `GiST` e `R-tree`.

## Notas

1. `dirty read` — A transação SQL `T1` altera uma linha. Em seguida a transação SQL `T2` lê esta linha antes de `T1` executar o comando `COMMIT`. Se depois `T1` executar o comando `ROLLBACK`, `T2` terá lido uma linha que nunca foi efetivada e que, portanto, pode ser considerada como nunca tendo existido. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
2. `nonrepeatable read` — A transação SQL `T1` lê uma linha. Em seguida a transação SQL `T2` altera ou exclui esta linha e executa o comando `COMMIT`. Se `T1` tentar ler esta linha novamente, pode receber o valor alterado ou descobrir que a linha foi excluída. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
3. `phantom read` — A transação SQL `T1` lê um conjunto de linhas `N` que satisfazem a uma condição de procura. Em seguida a transação SQL `T2` executa comandos SQL que geram uma ou mais linhas que satisfazem a condição de procura usada pela transação `T1`. Se depois a transação SQL `T1` repetir a leitura inicial com a mesma condição de procura, será obtida uma coleção diferente de linhas. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
4. Em essência, o sistema de bloqueio de predicado evita leituras fantasmas restringindo o que é escrito, enquanto o MVCC evita restringindo o que é lido.

# Capítulo 13. Dicas de desempenho

O desempenho dos comandos pode ser afetado por vários motivos. Alguns destes motivos podem ser tratados pelo usuário, enquanto outros são inerentes ao projeto do sistema subjacente. Este capítulo fornece algumas dicas para compreender e ajustar o desempenho do PostgreSQL.

## 13.1. Utilização do comando EXPLAIN

O PostgreSQL concebe um *plano de comando* para cada comando recebido. A escolha do plano correto, correspondendo à estrutura do comando e às propriedades dos dados, é absolutamente crítico para o bom desempenho. Pode ser utilizado o comando *EXPLAIN* para ver o plano criado pelo sistema para qualquer comando. A leitura do plano é uma arte que merece um tutorial extenso, o que este não é; porém, aqui são fornecidas algumas informações básicas.

Os números apresentados atualmente pelo *EXPLAIN* são:

- O custo de partida estimado (O tempo gasto antes de poder começar a varrer a saída como, por exemplo, o tempo para fazer a classificação em um nó de classificação).
- O custo total estimado (Se todas as linhas fossem buscadas, o que pode não acontecer: uma consulta contendo a cláusula *LIMIT* pára antes de gastar o custo total, por exemplo).
- Número de linhas de saída estimado para este nó do plano (Novamente, somente se for executado até o fim).
- Largura média estimada (em bytes) das linhas de saída deste nó do plano.

Os custos são medidos em termos de unidades de páginas de disco buscadas (O esforço de CPU estimado é convertido em unidades de páginas de disco, utilizando fatores estipulados altamente arbitrários. Se for desejado realizar experiências com estes fatores, consulte a lista de parâmetros de configuração em tempo de execução na Seção 16.4.5.2.)

É importante notar que o custo de um nó de nível mais alto inclui o custo de todos os seus nós descendentes. Também é importante perceber que o custo reflete apenas as coisas com as quais o planejador/otimizador se preocupa. Em particular, o custo não considera o tempo gasto transmitindo as linhas do resultado para o cliente, que pode ser o fator predominante no computo do tempo total gasto, mas que o planejador ignora porque não pode mudá-lo alterando o plano (Todo plano correto produz o mesmo conjunto de linhas, assim se acredita).

Linhas de saída é um pouco enganador, porque *não* é o número de linhas processadas/varridas pelo comando, geralmente é menos, refletindo a seletividade estimada de todas as condições da cláusula *WHERE* aplicadas a este nó. Idealmente, a estimativa de linhas do nível superior estará próxima do número de linhas realmente retornadas, atualizadas ou excluídas pelo comando.

Abaixo seguem alguns exemplos (utilizando o banco de dados de teste de regressão após a execução do comando *VACUUM ANALYZE*, e os códigos fonte de desenvolvimento da versão 7.3):

```
EXPLAIN SELECT * FROM tenk1;
```

```
              QUERY PLAN
-----
Seq Scan on tenk1  (cost=0.00..333.00 rows=10000 width=148)
```

Isto é tão direto quanto parece. Se for executado

```
SELECT * FROM pg_class WHERE relname = 'tenk1';
```

será visto que *tenk1* ocupa 233 páginas de disco e possui 10.000 linhas. Portanto, o custo é estimado em 233 páginas lidas, definidas como custando 1.0 cada uma, mais  $10.000 * \text{cpu\_tuple\_cost}$  que é atualmente 0.01 (execute *SHOW cpu\_tuple\_cost* para ver) ( $233 + 10.000 * 0,01 = 233 + 100 = 333$  - N. do T.).

Agora a consulta será modificada para incluir uma condição *WHERE*:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;
```

## QUERY PLAN

```
-----
Seq Scan on tenk1 (cost=0.00..358.00 rows=1033 width=148)
  Filter: (unique1 < 1000)
```

A estimativa de linhas de saída diminuiu por causa da cláusula `WHERE`. Entretanto, a varredura ainda precisa percorrer todas as 10.000 linhas e, portanto, o custo não diminuiu; na verdade aumentou um pouco, para refletir o tempo a mais de CPU gasto verificando a condição `WHERE`.

O número verdadeiro de linhas que esta consulta deveria selecionar é 1.000, mas a estimativa é somente aproximada. Se for tentado repetir esta experiência, provavelmente será obtida uma estimativa ligeiramente diferente; além disso, mudanças ocorrem após cada comando `ANALYZE`, porque as estatísticas produzidas pelo `ANALYZE` são obtidas a partir de amostras aleatórias na tabela.

Modificando-se a consulta para restringir mais ainda a condição

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50;
```

## QUERY PLAN

```
-----
Index Scan using tenk1_unique1 on tenk1 (cost=0.00..179.33 rows=49 width=148)
  Index Cond: (unique1 < 50)
```

será visto que quando fazemos a condição `WHERE` seletiva o bastante, o planejador decide, finalmente, que a varredura do índice tem custo menor que a varredura seqüencial. Este plano necessita buscar apenas 50 linhas por causa do índice e, portanto, vence apesar do fato de cada busca individual ser mais cara que a leitura de toda a página do disco seqüencialmente.

Adição de outra condição à cláusula `WHERE`:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50 AND string1 = 'xxx';
```

## QUERY PLAN

```
-----
Index Scan using tenk1_unique1 on tenk1 (cost=0.00..179.45 rows=1 width=148)
  Index Cond: (unique1 < 50)
  Filter: (string1 = 'xxx'::name)
```

A condição adicionada `string1 = 'xxx'` reduz a estimativa de linhas de saída mas não o custo, porque deverá ser acessado o mesmo conjunto de linhas. Deve ser observado que a cláusula `string1` não pode ser aplicada como uma condição do índice (porque o índice contém apenas a coluna `unique1`). Em vez disto, é aplicada como um filtro nas linhas trazidas pelo índice. Portanto, o custo na verdade sobe um pouco para refletir esta verificação adicional.

A seguir é feita a junção de duas tabelas, utilizando as colunas sendo discutidas:

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;
```

## QUERY PLAN

```
-----
Nested Loop (cost=0.00..327.02 rows=49 width=296)
-> Index Scan using tenk1_unique1 on tenk1 t1
      (cost=0.00..179.33 rows=49 width=148)
      Index Cond: (unique1 < 50)
-> Index Scan using tenk2_unique2 on tenk2 t2
      (cost=0.00..3.01 rows=1 width=148)
      Index Cond: ("outer".unique2 = t2.unique2)
```

Nesta junção de laço aninhado a varredura externa é a mesma varredura de índice vista no penúltimo exemplo e, portanto, seu custo e quantidade de linhas são os mesmos, porque está sendo aplicada a cláusula `unique1 < 50` neste nó. A cláusula `t1.unique2 = t2.unique2` ainda não é relevante e, portanto, não afeta a quantidade de linhas da varredura externa.

Para a varredura interna, o valor de `unique2` da linha da varredura externa corrente é vinculado à varredura interna do índice para produzir uma condição de índice como `t2.unique2 = constante`. Portanto, é obtido o mesmo plano e custo para a varredura interna que seria obtido por, digamos, `EXPLAIN SELECT * FROM tenk2 WHERE unique2 = 42`. Os custos do nó do laço são então definidos tomando por base o custo da varredura externa, mais uma repetição da varredura interna para cada linha externa ( $49 * 3.01$ , neste caso), mais um pouco de tempo de CPU para o processo de junção.

Neste exemplo, a quantidade de linhas de saída da junção é igual ao produto da quantidade de linhas das duas varreduras, mas isto usualmente não é verdade porque, em geral, podem existir cláusulas `WHERE` fazendo menção às duas tabelas e, portanto, só podem ser aplicadas no ponto de junção, e não às duas varreduras de entrada. Por exemplo, se fosse adicionado `WHERE ... AND t1.hundred < t2.hundred`, faria diminuir a quantidade de linhas de saída do nó da junção, mas não mudaria nenhuma das varreduras da entrada.

Uma forma de ver outros planos é forçar o planejador a não considerar a estratégia que sairia vencedora, habilitando e desabilitando sinalizadores de cada tipo de plano (Esta é uma ferramenta deselegante, mas útil. Consulte também a Seção 13.3).

```
SET enable_nestloop = off;
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;
```

QUERY PLAN

```
-----
Hash Join  (cost=179.45..563.06 rows=49 width=296)
  Hash Cond: ("outer".unique2 = "inner".unique2)
  -> Seq Scan on tenk2 t2  (cost=0.00..333.00 rows=10000 width=148)
  -> Hash  (cost=179.33..179.33 rows=49 width=148)
        -> Index Scan using tenk1_unique1 on tenk1 t1
              (cost=0.00..179.33 rows=49 width=148)
        Index Cond: (unique1 < 50)
```

Este plano propõe extrair as 50 linhas que interessam de `tenk1`, usando a mesma varredura de índice anterior, armazená-las para uso posterior em uma tabela de dispersão (hash table) em memória e, então, fazer uma varredura seqüencial em `tenk2` procurando possíveis correspondências na tabela de dispersão para `t1.unique2 = t2.unique2` para cada linha de `tenk2`. O custo para ler `tenk1` e montar a tabela de dispersão é inteiramente custo de partida para a junção hash, porque não haverá nenhuma linha de saída até começar a leitura de `tenk2`. O tempo total estimado para a junção também inclui uma pesada carga de tempo de CPU para verificar a tabela de dispersão 10.000 vezes. Entretanto, deve ser observado que *não* está sendo cobrado  $10.000 * 179,33$ ; a montagem da tabela de dispersão é feita somente uma vez neste tipo de plano.

É possível verificar a precisão dos custos estimados pelo planejador utilizando o comando `EXPLAIN ANALYZE`. Na verdade este comando executa a consulta, e depois mostra o tempo real acumulado dentro de cada nó do plano junto com os custos estimados que o comando `EXPLAIN` simples mostraria. Por exemplo, poderia ser obtido um resultado como este:

```
EXPLAIN ANALYZE SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;
```

QUERY PLAN

```
-----
Nested Loop  (cost=0.00..327.02 rows=49 width=296)
              (actual time=1.181..29.822 rows=50 loops=1)
  -> Index Scan using tenk1_unique1 on tenk1 t1
        (cost=0.00..179.33 rows=49 width=148)
        (actual time=0.630..8.917 rows=50 loops=1)
        Index Cond: (unique1 < 50)
  -> Index Scan using tenk2_unique2 on tenk2 t2
        (cost=0.00..3.01 rows=1 width=148)
        (actual time=0.295..0.324 rows=1 loops=50)
        Index Cond: ("outer".unique2 = t2.unique2)
Total runtime: 31.604 ms
```



Deve ser observado que os valores de “actual time” são em milissegundos de tempo real, enquanto as estimativas de “custo” (*cost*) são expressas em unidades arbitrárias de busca em disco; portanto, não é provável haver correspondência. É nas relações que se deve prestar atenção.

Em alguns planos de comando é possível que um nó de subplano seja executado mais de uma vez. Por exemplo, a varredura de índice interna é executada uma vez para cada linha externa no plano de laço aninhado acima. Nestes casos, o valor “loops” (laços) expressa o número total de execuções do nó, e os valores *actual time* (tempo real) e *rows* (linhas) mostrados são valores médios por execução. Isto é feito para tornar os números comparáveis com o modo como as estimativas de custo são mostradas. Deve ser multiplicado pelo valor “loops” para obter o tempo total realmente gasto no nó.

O *Total runtime* (tempo total de execução) mostrado pelo *EXPLAIN ANALYZE* inclui os tempos de inicialização e de finalização do executor, assim como o tempo gasto processando as linhas do resultado. Não inclui os tempos de análise, reescrita e planejamento. Para um comando *SELECT*, o tempo total de execução normalmente será apenas um pouco maior que o tempo total informado para o nó do plano de nível mais alto. Para os comandos *INSERT*, *UPDATE* e *DELETE*, o tempo total de execução pode ser consideravelmente maior, porque inclui o tempo gasto processando as linhas do resultado. Nestes comandos, o tempo para o nó superior do plano é, essencialmente, o tempo gasto computando as novas linhas e/ou localizando as linhas antigas, mas não inclui o tempo gasto realizando as alterações. O tempo gasto disparando os gatilhos, se houver algum, está fora do nó superior do plano, sendo mostrado separadamente para cada gatilho.

Vale a pena notar que os resultados do comando *EXPLAIN* não devem ser extrapolados para outras situações além da que está sendo testada; por exemplo, não é possível supor que os resultados para uma tabela pequena possam ser aplicados a uma tabela grande. As estimativas de custo do planejador não são lineares e, portanto, podem ser escolhidos planos diferentes para tabelas maiores ou menores. Um exemplo extremo é o de uma tabela que ocupa uma única página em disco, onde quase sempre vence o plano de varredura seqüencial, havendo índices disponíveis ou não. O planejador percebe que fará a leitura de uma página do disco para processar a tabela em qualquer caso e, portanto, não faz sentido fazer leituras de páginas adicionais para procurar em um índice.

## 13.2. Estatísticas utilizadas pelo planejador

Conforme visto na seção anterior, o planejador de comandos precisa estimar o número de linhas buscadas pelo comando para poder fazer boas escolhas dos planos de comando. Esta seção fornece uma rápida visão das estatísticas utilizadas pelo sistema para fazer estas estimativas.

Um dos componentes da estatística é o número total de entradas em cada tabela e índice, assim como o número de blocos de disco ocupados por cada tabela e índice. Esta informação é mantida nas colunas *reltuples* e *relpages* da tabela *pg\_class*, podendo ser vista utilizando consultas semelhantes à mostrada abaixo:

```
SELECT relname, relkind, reltuples, relpages FROM pg_class WHERE relname LIKE 'tenk1%';
```

relname	relkind	reltuples	relpages
tenk1	r	10000	233
tenk1_hundred	i	10000	30
tenk1_unique1	i	10000	30
tenk1_unique2	i	10000	30

(4 linhas)

Pode ser visto que *tenk1* contém 10.000 linhas, assim como seus índices, mas que os índices são (sem surpresa) muito menores que a tabela.

Por razões de eficiência, as colunas *reltuples* e *relpages* não são atualizadas dinamicamente e, portanto, usualmente contêm valores um pouco desatualizados. São atualizadas pelos comandos *VACUUM*, *ANALYZE* e uns poucos comandos de DDL como *CREATE INDEX*. Um comando *ANALYZE* autônomo, ou seja, não fazendo parte do *VACUUM*, gera um valor aproximado para *reltuples* uma vez que não lê todas as linhas da tabela. O planejador faz uma proporcionalidade dos valores encontrados em *pg\_class* para que correspondam ao tamanho físico corrente da tabela, obtendo assim uma estimativa mais próxima.

A maioria dos comandos busca apenas uma fração das linhas da tabela, porque possuem cláusulas *WHERE* que restringem as linhas a serem examinadas. Portanto, o planejador precisa fazer uma estimativa da *seletividade* das cláusulas *WHERE*, ou seja, a fração das linhas correspondendo a cada condição na cláusula *WHERE*. A informação utilizada para esta tarefa é

armazenada no catálogo do sistema `pg_statistic`. As entradas em `pg_statistic` são atualizadas pelos comandos `ANALYZE` e `VACUUM ANALYZE`, sendo sempre aproximadas, mesmo logo após serem atualizadas.

Em vez de olhar diretamente em `pg_statistic`, é melhor olhar sua visão `pg_stats` ao se examinar as estatísticas manualmente. A visão `pg_stats` foi projetada para ser lida mais facilmente. Além disso, `pg_stats` pode ser lida por todos, enquanto `pg_statistic` somente pode ser lida pelos superusuários (Isto impede que usuários não privilegiados aprendam algo sobre o conteúdo das tabelas de outras pessoas a partir de suas estatísticas. A visão `pg_stats` tem como restrição mostrar somente informações sobre as tabelas que o usuário corrente pode ler). Por exemplo, podemos executar:

```
SELECT attname, n_distinct, most_common_vals FROM pg_stats WHERE tablename = 'road';
```

```
attname | n_distinct |          most_common_vals
-----+-----+-----
name    |    -0.467008 | { "I- 580                      Ramp", ... }
thepath |           20 | { " [ (-122.089,37.71) , (-122.0886,37.711) ] " }
(2 linhas)
```

```
/*
 * onde ... representa:
 * "I- 880                      Ramp",
 * "Sp Railroad                  ",
 * "I- 580                      ",
 * "I- 680                      Ramp",
 * "I- 80                       Ramp",
 * "14th                        St  ",
 * "5th                         St  ",
 * "Mission                     Blvd",
 * "I- 880                      "
 */
```

A visão `pg_stats` está descrita detalhadamente na Seção 41.36.

A quantidade de informação armazenada em `pg_statistic`, em particular o número máximo de entradas nas matrizes `most_common_vals` e `histogram_bounds` para cada coluna, podem ser definidas coluna por coluna utilizando o comando `ALTER TABLE SET STATISTICS`, ou globalmente definindo a variável de configuração `default_statistics_target`. Atualmente o limite padrão são 10 entradas. Aumentar o limite pode permitir que o planejador faça estimativas mais precisas, particularmente para colunas com distribuição irregular dos dados, porém consumindo mais espaço na tabela `pg_statistic` e um pouco mais de tempo para computar as estimativas. Inversamente, um limite mais baixo pode ser apropriado para colunas com distribuição dos dados simples.

### 13.3. Controle do planejador com cláusulas JOIN explícitas

É possível ter algum controle sobre o planejador de comandos utilizando a sintaxe `JOIN` explícita. Para saber por que isto tem importância, primeiro é necessário ter algum conhecimento.

Em uma consulta de junção simples, como

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

o planejador está livre para fazer a junção das tabelas em qualquer ordem. Por exemplo, pode gerar um plano de comando que faz a junção de A com B utilizando a condição `a.id = b.id` do `WHERE` e, depois, fazer a junção de C com a tabela juntada utilizando a outra condição do `WHERE`. Poderia, também, fazer a junção de B com C e depois juntar A ao resultado. Ou, também, fazer a junção de A com C e depois juntar B, mas isto não seria eficiente, uma vez que deveria ser produzido o produto Cartesiano completo de A com C, porque não existe nenhuma condição aplicável na cláusula `WHERE` que permita a otimização desta junção (Todas as junções no executor do PostgreSQL acontecem entre duas tabelas de entrada sendo, portanto, necessário construir o resultado a partir de uma ou outra destas formas). O ponto a ser destacado é que estas diferentes possibilidades de junção produzem resultados semanticamente equivalentes, mas podem ter custos de execução muito diferentes. Portanto, o planejador explora todas as possibilidades tentando encontrar o plano de consulta mais eficiente.

Quando uma consulta envolve apenas duas ou três tabelas não existem muitas ordens de junção com que se preocupar. Porém, o número de ordens de junção possíveis cresce exponencialmente quando o número de tabelas aumenta. Acima de

dez tabelas de entrada não é mais prático efetuar uma procura exaustiva por todas as possibilidades, e mesmo para seis ou sete tabelas o planejamento pode levar um longo tempo. Quando existem muitas tabelas de entrada, o planejador do PostgreSQL pode alternar da procura exaustiva para a procura probabilística *genética* através de um número limitado de possibilidades (O ponto de mudança é definido pelo parâmetro em tempo de execução `geqo_threshold`). A procura genética leva menos tempo, mas não encontra necessariamente o melhor plano possível.

Quando a consulta envolve junções externas, o planejador tem muito menos liberdade que com as junções puras (internas). Por exemplo, considere:

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Embora as restrições desta consulta sejam superficialmente semelhantes às do exemplo anterior as semânticas são diferentes, porque deve ser gerada uma linha para cada linha de A que não possua linha correspondente na junção de B com C. Portanto, o planejador não pode escolher a ordem de junção neste caso: deve fazer a junção de B com C e depois juntar A ao resultado. Portanto, esta consulta leva menos tempo para ser planejada que a consulta anterior.

A sintaxe de junção interna explícita (`INNER JOIN`, `CROSS JOIN` ou `JOIN` sem adornos) é semanticamente idêntica a listar as relações de entrada na cláusula `FROM`, portanto não precisa restringir a ordem de junção. Ainda assim é possível instruir o planejador de comandos do PostgreSQL para que trate os `JOINS` internos explícitos como restringindo a ordem de junção. Por exemplo, estas três consultas são logicamente equivalentes:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Mas se dissermos ao planejador para que respeite a ordem do `JOIN`, a segunda e a terceira consultas levam menos tempo para serem planejadas que a primeira. Não vale a pena se preocupar com este efeito para apenas três tabelas, mas pode ser de grande valia para muitas tabelas.

Para obrigar o planejador seguir a ordem do `JOIN` para as junções internas, deve ser definido o parâmetro em tempo de execução `join_collapse_limit` como 1 (São mostrados abaixo outros valores possíveis).

Não é necessário restringir completamente a ordem de junção para diminuir o tempo de procura, porque podem ser utilizados operadores `JOIN` entre os itens da lista `FROM` pura. Por exemplo,

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

Com `join_collapse_limit = 1` isto obriga o planejador a fazer a junção de A com B antes de juntá-las às outras tabelas, mas não restringe suas escolhas além disso. Neste exemplo, o número de ordens de junção possíveis é reduzido por um fator de 5.

Restringir a procura do planejador desta maneira é uma técnica útil tanto para reduzir o tempo de planejamento quanto para direcionar o planejador para um bom plano de comando. Se o planejador escolher uma ordem de junção ruim por padrão, é possível forçá-lo a escolher uma ordem melhor por meio da sintaxe do `JOIN` — assumindo que se conheça uma ordem melhor. Recomenda-se realizar experiências.

Uma questão intimamente relacionada, que afeta o tempo de planejamento, é o colapso das subconsultas dentro da consulta ancestral. Por exemplo, considere

```
SELECT *
FROM x, y,
      (SELECT * FROM a, b, c WHERE alguma_coisa) AS ss
WHERE uma_outra_coisa;
```

Esta situação pode ocorrer quando se utiliza uma visão contendo uma junção; a regra `SELECT` da visão é inserida no lugar da referência à visão, produzindo uma consulta muito parecida com a mostrada acima. Normalmente, o planejador tenta fazer o colapso da subconsulta dentro da consulta externa, produzindo

```
SELECT * FROM x, y, a, b, c WHERE alguma_coisa AND uma_outra_coisa;
```

Geralmente isto resulta em um plano melhor que planejar a subconsulta separadamente (Por exemplo, as condições do `WHERE` externo podem ser tais que a junção de X com A primeiro elimine muitas linhas de A evitando, portanto, a necessidade de formar a saída lógica completa da subconsulta). Mas ao mesmo tempo foi aumentado o tempo de

planejamento; agora temos um problema de junção de cinco vias no lugar de dois problemas de junção de três vias separados. Devido ao crescimento exponencial do número de possibilidades, isto faz uma grande diferença. O planejador tenta evitar ficar preso a problemas de procura de junção grande não fazendo o colapso da subconsulta se resultar em mais que `from_collapse_limit` itens na cláusula `FROM` da consulta externa. É possível equilibrar o tempo de planejamento versus a qualidade do plano ajustando este parâmetro de tempo de execução para cima ou para baixo.

`from_collapse_limit` e `join_collapse_limit` possuem nomes semelhantes porque fazem praticamente a mesma coisa: um controla quando o planejador irá “aplanar” (`flatten out`) as subconsultas, e o outro controla quando serão aplanadas as junções internas explícitas. Normalmente `join_collapse_limit` é definido igual a `from_collapse_limit` (fazendo as junções explícitas e as subconsultas agirem da mesma maneira), ou `join_collapse_limit` é definido igual a 1 (se for desejado controlar a ordem de junção com junções explícitas). Mas podem ser definidas com valores diferentes quando se tenta aprimorar o equilíbrio entre o tempo de planejamento e o tempo de execução.

## 13.4. Carga dos dados no banco

Pode ser necessário inserir uma grande quantidade de dados ao se fazer a carga inicial do banco de dado. Esta seção contém algumas sugestões sobre como tornar este processo tão eficiente quanto possível.

### 13.4.1. Desabilitar a efetivação automática

Desabilitar a efetivação automática (`autocommit`) e fazer apenas uma efetivação no final (Em puro SQL, isto significa executar `BEGIN` no começo e `COMMIT` no fim. Algumas bibliotecas cliente podem fazer isto às escondidas e, neste caso, é preciso ter certeza que a biblioteca só faz quando se deseja que seja feito). Se permitirmos efetivar cada inserção separadamente, o PostgreSQL terá muito trabalho para cada linha inserida. Um benefício adicional de fazer todas as inserções em uma transação, é que se a inserção de uma linha não for bem-sucedida, então a inserção de todas as linhas feita até este ponto é desfeita, não sendo necessário se preocupar com a carga parcial dos dados.

### 13.4.2. Uso do COPY FROM

Usar o comando `COPY` para carregar todas as linhas em um comando, em vez de usar uma série de comandos `INSERT`. O comando `COPY` é otimizado para carregar uma grande quantidade de linhas; é menos flexível que o comando `INSERT`, mas ocasiona uma sobrecarga significativamente menor para cargas de dados volumosas. Uma vez que o comando `COPY` é um único comando, não é necessário desabilitar a auto-efetivação se for utilizado este método para carregar a tabela.

Se não for possível utilizar o comando `COPY`, pode ser útil utilizar o comando `PREPARE` para criar comandos `INSERT` preparados, e depois usar o comando `EXECUTE` tantas vezes quanto forem necessárias. Isto evita a sobrecarga de analisar e planejar o comando `INSERT` repetidas vezes.

Deve-se notar que carregar uma grande quantidade de linhas utilizando o comando `COPY` é quase sempre mais rápido que utilizando o comando `INSERT`, mesmo que o comando `PREPARE` seja utilizado e sejam feitas várias inserções em lote em uma única transação.

### 13.4.3. Remoção dos índices

Se estiver sendo carregada uma tabela recém criada, a maneira mais rápida é criar a tabela, carregar os dados usando o `COPY` e, depois, criar os índices necessários para a tabela. Criar um índice sobre dados pré-existent é mais rápido que atualizá-lo de forma incremental durante a carga de cada linha.

Para aumentar uma tabela existente, pode-se remover o índice, carregar a tabela e, depois, recriar o índice. É claro que o desempenho do banco de dados para os outros usuários será afetado negativamente durante o tempo que o índice não existir. Deve-se pensar duas vezes antes de remover um índice único, porque a verificação de erro efetuada pela restrição de unicidade não existirá enquanto o índice não tiver sido criado novamente.

### 13.4.4. Aumento de maintenance\_work\_mem

O aumento temporário da variável de configuração `maintenance_work_mem` durante a carga de uma grande quantidade de dados pode ocasionar uma melhora no desempenho. Isto se deve ao fato de quando o índice `B-tree` é criado a partir do início, o conteúdo presente na tabela precisa ser ordenado. Permitir o `merge sort`<sup>1</sup> utilizar mais memória significa que serão necessários menos passos de mesclagem. Uma definição maior para `maintenance_work_mem` também pode acelerar a validação de restrições de chave estrangeira.

### 13.4.5. Aumento de checkpoint\_segments

O aumento temporário da variável de configuração `checkpoint_segments` também pode tornar as cargas de dados volumosas mais rápidas. Isto se deve ao fato de quando uma grande quantidade de dados é carregada no PostgreSQL, isto pode fazer com que os pontos de controle (`checkpoints`) ocorram com mais frequência que a frequência normal de ponto de controle (especificada pela variável de configuração `checkpoint_timeout`). Sempre que ocorre um ponto de controle, todas as páginas sujas (`dirty pages`) devem ser descarregadas no disco. Aumentado-se `checkpoint_segments` temporariamente durante uma carga volumosa, o número de pontos de controle necessários pode ser menor.

### 13.4.6. Depois executar o comando ANALYZE

Após se alterar significativamente a distribuição dos dados da tabela, é recomendado executar o comando *ANALYZE*. Isto inclui a carga de grande quantidade de dados na tabela. A execução do comando *ANALYZE* (ou *VACUUM ANALYZE*) garante que o planejador possuirá estatísticas atualizadas sobre a tabela. Sem estatísticas, ou com estatísticas obsoletas, o planejador pode tomar decisões ineficientes durante o planejamento dos comandos, ocasionando um desempenho fraco nas tabelas com estatísticas imprecisas ou não existentes.

## Notas

1. `merge sort` — Um algoritmo de classificação que divide os itens a serem classificados em dois grupos, classifica cada grupo recursivamente e, depois, mescla os grupos em uma sequência classificada final. National Institute of Standards and Technology (<http://www.nist.gov/dads/HTML/mergesort.html>) (N. do T.)

# III. Administração do servidor

Esta parte cobre tópicos que são de interesse do administrador de banco de dados do PostgreSQL. Inclui a instalação do produto, configuração do servidor, gerenciamento de usuários e de bancos de dados, e tarefas de manutenção. Todos que gerenciam um servidor PostgreSQL para uso pessoal ou, especialmente, de produção, devem estar familiarizados com os tópicos cobertos nesta parte.

As informações estão organizadas, aproximadamente, na ordem pela qual um novo usuário deve lê-las. Porém, os capítulos são auto-contidos podendo ser lidos individualmente conforme desejado. As informações estão apresentadas sob forma narrativa, sendo cada unidade um tópico. Os leitores à procura de uma descrição completa de um determinado comando devem consultar a Parte VI.

Os capítulos iniciais estão escritos de forma que possam ser entendidos sem pré-requisitos de conhecimento e, portanto, os novos usuários com necessidade de instalar seus próprios servidores podem começar a leitura por estes capítulos. O restante está relacionado com ajuste e gerenciamento, pressupondo que o leitor esteja familiarizado com o uso geral do sistema de banco de dados PostgreSQL. Incentivamos os leitores lerem a Parte I e a Parte II para obter informações adicionais.

# Capítulo 14. Instruções de instalação

Este capítulo descreve a instalação do PostgreSQL a partir do código fonte da distribuição (Se estiver sendo instalada uma distribuição pré-empacotada, como RPM ou um pacote Debian, este capítulo deve ser ignorado, e em seu lugar devem ser lidas as instruções do pacote).

## 14.1. Resumo da instalação

```
./configure
gmake
su
gmake install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data >logfile 2>&1 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

A versão integral é o restante deste capítulo.

## 14.2. Requisitos

De uma maneira geral, uma plataforma atual compatível com o Unix deve ser capaz de executar o PostgreSQL. Na Seção 14.7 estão relacionadas as plataformas onde foram feitos testes específicos até o momento em que esta versão foi liberada. No subdiretório doc da distribuição do código fonte existem diversos documentos FAQ específicos de plataforma, que podem ser consultados em caso de problema.

São necessários os seguintes pacotes de software para construir o PostgreSQL:

- É requerido o make do GNU; outros programas make *não* funcionam. O make do GNU está instalado geralmente sob o nome gmake; este documento sempre fará referência ao make do GNU utilizando este nome (Em alguns outros sistemas o make do GNU é a ferramenta padrão com o nome make<sup>1</sup>). Para testar o make do GNU deve ser executado

```
gmake --version
```

```
# No Fedora Core 3 (N. do T.)
```

```
make --version
```

```
GNU Make 3.80
Copyright (C) 2002 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
```

Recomenda-se a utilização da versão 3.76.1 ou mais recente.

- É necessário um compilador C ISO/ANSI. Recomenda-se as versões recentes do GCC, mas o PostgreSQL é conhecido por poder ser construído por uma grande variedade de compiladores de diferentes fornecedores.
- Antes de tudo é necessário o gzip para descompactar a distribuição.
- A biblioteca Readline do GNU (para editar linhas confortavelmente e recuperar o histórico de comandos) é utilizada por padrão. Se não for desejado usá-la então é necessário especificar a opção `--without-readline` no configure (No NetBSD a biblioteca `libedit` é compatível com o Readline, sendo utilizada se `libreadline` não for encontrada). Se estiver sendo utilizada uma distribuição baseada no Linux, esteja ciente que é necessário tanto o pacote `readline` quanto o pacote `readline-devel`, caso sejam pacotes separados na distribuição sendo utilizada.

- Para construir o PostgreSQL no Windows, é necessário software adicional. O PostgreSQL pode ser construído nas versões do Windows baseadas no NT, como o Windows 2000, XP ou 2003, utilizando o MinGW; para obter detalhes deve ser visto o arquivo `doc/FAQ_MINGW` na distribuição do código fonte. O PostgreSQL também pode ser construído utilizando o Cygwin; para obter detalhes deve ser visto o arquivo `doc/FAQ_CYGWIN` na distribuição do código fonte. As construções baseadas no Cygwin funcionam nas versões do Windows mais antigas mas, se houver possibilidade de escolha, recomenda-se utilizar o MinGW. Embora estes sejam os únicos conjuntos de ferramentas recomendados para uma construção completa, é possível construir apenas a biblioteca C cliente (`libpq`) e o terminal interativo (`psql`) utilizando outros conjuntos de ferramentas do Windows. Para obter detalhes deve ser visto o Capítulo 15.

Os pacotes a seguir são opcionais. Embora não sejam requeridos pela configuração padrão, são necessários quando certas opções de construção são habilitadas, conforme explicado abaixo.

- Para construir a linguagem de programação do servidor PL/Perl é necessária uma instalação completa do Perl, incluindo a biblioteca `libperl` e os arquivos de cabeçalho. Uma vez que o PL/Perl será uma biblioteca compartilhada, a biblioteca `libperl` também deve ser uma biblioteca compartilhada na maioria das plataformas. Isto parece ser o padrão nas versões recentes do Perl, mas não era nas versões mais antigas e, de qualquer maneira, foi uma escolha de quem instalou o Perl no sistema.

Se não houver a biblioteca compartilhada, mas esta for necessária, uma mensagem como a mostrada abaixo será exibida durante a construção para chamar atenção sobre este fato:

```
*** Cannot build PL/Perl because libperl is not a shared library.
*** You might have to rebuild your Perl installation. Refer to
*** the documentation for details.
```

(Se a saída na tela não for acompanhada, apenas será observado que a biblioteca objeto PL/Perl, `plperl.so` ou semelhante, não foi instalada). Se esta mensagem for vista, será necessário reconstruir e instalar o Perl manualmente para poder construir o PL/Perl. Durante o processo de configuração do Perl deve ser requerida a geração da biblioteca compartilhada.

- Para construir a linguagem de programação do servidor PL/Python é necessária a instalação do Python, incluindo os arquivos de cabeçalho e o módulo `distutils`. O módulo `distutils` é incluído por padrão pelo Python 1.6 e posteriores; os usuários das versões mais antigas do Python terão que instalá-lo.

Uma vez que o PL/Python será uma biblioteca compartilhada, a biblioteca `libpython` também deve ser uma biblioteca compartilhada na maioria das plataformas. Este não é o caso em uma instalação padrão do Python. Se após a construção e instalação houver um arquivo chamado `plpython.so` (possivelmente com uma extensão diferente), então tudo correu bem. Senão, deve ter sido exibida uma mensagem como esta na tela:

```
*** Cannot build PL/Python because libpython is not a shared library.
*** You might have to rebuild your Python installation. Refer to
*** the documentation for details.
```

Isto significa que é necessário reconstruir (parte) da instalação do Python, para ser gerada a biblioteca compartilhada.

Caso ocorram problemas, deve ser executada a configuração do Python 2.3, ou mais recente, utilizando o sinalizador `--enable-shared`. Em alguns sistemas operacionais não é necessário construir a biblioteca compartilhada, mas é necessário convencer o construtor do sistema PostgreSQL disso. Para obter detalhes deve ser consultado o arquivo `Makefile` no diretório `src/pl/plpython`.

- Se for desejado construir a linguagem procedural PL/Tcl, obviamente é necessário instalar o Tcl.
- Para habilitar o Suporte ao Idioma Nativo (Native Language Support - NLS), ou seja, a capacidade de mostrar as mensagens do programa em um idioma diferente do Inglês, é necessária a implementação da API Gettext. Alguns sistemas operacionais possuem nativamente (por exemplo, Linux, NetBSD e Solaris), para outros sistemas pode ser baixado um pacote adicional de <http://developer.postgresql.org/~petere/bsd-gettext/>. Se estiver sendo utilizada a implementação do Gettext da biblioteca C GNU, então há a necessidade adicional do pacote GNU Gettext por alguns programas utilitários. Para nenhuma outra implementação isto é necessário.
- Kerberos, OpenSSL ou PAM, se for desejado permitir autenticação ou criptografia utilizando um destes serviços.

Se a construção estiver sendo feita a partir da árvore do CVS em vez do pacote fonte liberado, ou se for desejado fazer desenvolvimento, também são necessários os seguintes pacotes:



- O Flex e o Bison do GNU são necessários para construir a saída (*checkout*) do CVS, ou se forem alterados os arquivos de definição do varredor e do analisador existentes. Se forem necessários, certifique-se de trazer o Flex 2.5.4 ou mais recente, e o Bison 1.875 ou mais recente. Outros programas yacc podem, às vezes, serem utilizados, mas para se fazer isto é necessário um trabalho extra, e não é recomendado. Outros programas lex com certeza não funcionam.

Se for necessário obter um pacote GNU, este pode ser encontrado em um espelho do GNU (veja a relação em <http://www.gnu.org/order/ftp.html>) ou em <ftp://ftp.gnu.org/gnu/>.

Verifique, também, se há espaço suficiente no disco. São necessários cerca de 65 MB para a árvore de fontes durante a compilação, e cerca de 15 MB para o diretório de instalação. Um agrupamento de banco de dados vazio ocupa cerca de 25 MB, e os bancos de dados ocupam cerca de cinco vezes a quantidade de espaço ocupada pelos arquivos texto puro contendo os mesmos dados. Se forem executados os testes de regressão serão necessários, temporariamente, mais 90 MB. Para verificar o espaço em disco deve ser utilizado o comando `df`.

### 14.3. Obtenção dos arquivos fonte

Os fontes do PostgreSQL 8.0.0 podem ser obtidos por FTP anônimo a partir de <ftp://ftp.postgresql.org/pub/source>. Se for possível, deve ser utilizado um espelho. Após obter o arquivo, este deve ser descompactado:

```
gunzip postgresql-8.0.0.tar.gz
tar xf postgresql-8.0.0.tar
```

Este procedimento cria o diretório `postgresql-8.0.0`, sob o diretório corrente, contendo os fontes do PostgreSQL. Este diretório deve ser tornado o diretório corrente para efetuar o restante do procedimento de instalação.

### 14.4. Se estiver atualizando

O formato interno de armazenamento dos dados muda com as novas versões do PostgreSQL. Portanto, se estiver sendo feita a atualização de uma instalação existente, que não possua um número de versão “8.0.x”, é necessário fazer uma cópia de segurança e restaurar os dados conforme mostrado. Estas instruções assumem que a instalação existente se encontra sob o diretório `/usr/local/pgsql`, e que a área de dados está no diretório `/usr/local/pgsql/data`. Os caminhos devem ser modificados conforme seja apropriado.

1. Certifique-se que o banco de dados não seja atualizado durante ou após a realização da cópia de segurança. Isto não afeta a integridade da cópia de segurança, mas os dados modificados, obviamente, não serão incluídos. Se for necessário, devem ser editadas as permissões no arquivo `/usr/local/pgsql/data/pg_hba.conf`, ou equivalente, para não permitir acesso pelos demais usuários.
2. Para realizar a cópia de segurança da instalação de banco de dados deve-se executar:

```
pg_dumpall > arquivo_de_saida
```

Se for necessário preservar os OIDs (como quando são utilizados como chaves estrangeiras), então deve ser utilizada a opção `-o` ao se executar o `pg_dumpall`.

O `pg_dumpall` não salva os objetos grandes. Se for necessário salvá-los, deve ser consultada a Seção 22.1.4.

Para fazer a cópia de segurança, pode ser utilizado o utilitário `pg_dumpall` da versão sendo usada atualmente. Entretanto, para obter melhores resultados, deve-se tentar utilizar o utilitário `pg_dumpall` do PostgreSQL 8.0.0, uma vez que esta versão contém correções de erros e melhorias com relação às versões mais antigas. Embora este conselho possa parecer sem sentido, uma vez que a nova versão ainda não foi instalada, é aconselhável segui-lo se estiver sendo planejado instalar a nova versão em paralelo com a versão antiga. Neste caso pode-se completar a instalação normalmente, e transferir os dados depois. Isto também pode diminuir o tempo de máquina parada.

3. Se a nova versão estiver sendo instalada no mesmo local da versão antiga, então o servidor antigo deve ser parado antes de instalar os novos arquivos:

```
pg_ctl stop
```

Nos sistemas onde o PostgreSQL é ativado quando a máquina é ligada, provavelmente existe um arquivo de inicialização para obter o mesmo resultado. Por exemplo, no sistema operacional Red Hat Linux pode-se verificar que

```
/etc/rc.d/init.d/postgresql stop
```

funciona.

As versões muito antigas podem não ter o utilitário `pg_ctl`. Se este não puder ser encontrado, ou se não funcionar, o ID de processo do servidor antigo deve ser descoberto utilizando, por exemplo,

```
ps ax | grep postmaster
```

e depois ser enviado um sinal para parar o processo servidor desta forma:

```
kill -INT ID_processo
```

4. Se estiver sendo feita a instalação no mesmo local da versão antiga, então também é uma boa idéia mudar de lugar a instalação antiga, para o caso de haver problema e ser necessário voltar atrás. Deve ser usado um comando como este:

```
mv /usr/local/pgsql /usr/local/pgsql.antigo
```

Após ter sido feita a instalação do PostgreSQL 8.0.0, deve ser criado o novo diretório de banco de dados e ativado o novo servidor. Lembre-se que é necessário executar estes comandos através de uma conta especial de usuário do banco de dados (que já existe se estiver sendo feita uma atualização).

```
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
```

Por fim, os dados são restaurados pelo comando

```
/usr/local/pgsql/bin/psql -d template1 -f arquivo_de_saida
```

usando o *novo* psql.

Estes tópicos são mostrados na Seção 22.4, que incentivamos sua leitura em todos os casos.

## 14.5. Procedimento de instalação

1. Configuração:

O primeiro passo do procedimento de instalação é configurar a árvore de fontes do sistema a ser construído, e escolher as opções desejadas. Isto é feito executando o script `configure`. Para uma instalação padrão deve-se simplesmente executar:

```
./configure
```

Este script executa vários testes para obter valores para diversas variáveis dependentes do sistema e detectar comportamentos adversos do sistema operacional e, finalmente, cria vários arquivos na árvore de construção para registrar o que foi encontrado (O `configure` também pode ser executado em um diretório fora da árvore de fontes, se for desejado manter o diretório de construção em separado).

A configuração padrão constrói o servidor e os utilitários, assim como todos os aplicativos cliente e interfaces que requerem somente o compilador C. Todos os arquivos são instalados sob `/usr/local/pgsql` por padrão.

O processo de construção e instalação pode ser personalizado fornecendo uma ou mais das seguintes opções de linha de comando para o `configure`:

```
--prefix=PREFIXO
```

Instala todos os arquivos sob o diretório `PREFIXO` em vez de `/usr/local/pgsql`. Os arquivos são instalados em vários subdiretórios; nenhum arquivo é instalado diretamente no diretório `PREFIXO`.

Havendo necessidades especiais, os subdiretórios podem ser personalizados individualmente através das opções que se seguem. Entretanto, se forem deixados com o valor padrão, a instalação será movível, significando que o diretório poderá ser movido para outro local após a instalação (Os locais `man` e `doc` não são afetados por este procedimento).

Para as instalações movíveis, é necessário utilizar a opção `--disable-rpath` do `configure`. Também é necessário informar ao sistema operacional como encontrar as bibliotecas compartilhadas.

`--exec-prefix=EXEC-PREFIXO`

Os arquivos dependentes da arquitetura podem ser instalados sob um prefixo diferente (*EXEC-PREFIXO*) do que foi definido em *PREFIXO*, o que pode ser útil para compartilhar entre hospedeiros arquivos não dependentes de arquitetura. Se for omitido, então *EXEC-PREFIXO* é definido igual a *PREFIXO*, e tanto os arquivos dependentes da arquitetura quanto os que não dependem da arquitetura são instalados sob a mesma árvore, que provavelmente é o que se deseja.

`--bindir=DIRETÓRIO`

Especifica o diretório dos programas executáveis. O padrão é *EXEC-PREFIXO/bin*, o que normalmente corresponde a */usr/local/pgsql/bin*.

`--datadir=DIRETÓRIO`

Define o diretório dos arquivos somente para leitura utilizados pelos programas instalados. O padrão é *PREFIXO/share*. Deve ser observado que isto não tem nada a ver com o local onde os arquivos do banco de dados serão colocados.

`--sysconfdir=DIRETÓRIO`

O diretório de vários arquivos de configuração. O padrão é *PREFIXO/etc*.

`--libdir=DIRETÓRIO`

O local para instalar as bibliotecas e os módulos carregáveis dinamicamente. O padrão é *EXEC-PREFIXO/lib*.

`--includedir=DIRETÓRIO`

O diretório para instalar os arquivos de cabeçalho das linguagens C e C++. O padrão é *PREFIXO/include*.

`--mandir=DIRETÓRIO`

As páginas de manual (*man pages*) que acompanham o PostgreSQL são instaladas sob este diretório, nos respectivos subdiretórios *manx*. O padrão é *PREFIXO/man*.

`--with-docdir=DIRETÓRIO`

`--without-docdir`

Os arquivos da documentação, exceto as “man pages”, são instalados neste diretório. O padrão é *PREFIXO/doc*. Se for especificada a opção `--without-docdir`, a documentação não será instalada por `make install`. Isto tem por finalidade atender scripts de empacotamento que possuem métodos especiais para instalar a documentação.

**Nota:** Tomou-se cuidado para tornar possível a instalação do PostgreSQL em locais de instalação compartilhados (tal como */usr/local/include*) sem interferir com o espaço de nomes do restante do sistema. Primeiro, é anexada automaticamente a cadeia de caracteres “/pgsql” a *datadir*, *sysconfdir* e *docdir*, a menos que o nome do diretório inteiramente expandido contenha a cadeia de caracteres “postgres” ou “pgsql”. Por exemplo, se for escolhido */usr/local* como prefixo, a documentação será instalada em */usr/local/doc/postgresql*, mas se o prefixo for */opt/postgres*, então será instalada em */opt/postgres/doc*. Os arquivos de cabeçalho C públicos das interfaces cliente são instalados em *includedir* e são livres de espaço de nomes. Os arquivos de cabeçalho internos e os arquivos de cabeçalho do servidor são instalados em diretórios privativos sob *includedir*. Deve ser vista a documentação de cada interface para obter informações sobre como encontrar seus respectivos arquivos de cabeçalho. Finalmente, também é criado um subdiretório privativo, se for apropriado, sob *libdir*, para os módulos carregáveis dinamicamente.

`--with-includes=DIRETÓRIOS`

*DIRETÓRIOS* é uma lista de diretórios, separados por dois-pontos (:), a ser adicionada à lista usada pelo compilador para procurar por arquivos de cabeçalho. Havendo pacotes opcionais (como o Readline do GNU), instalados em locais diferente do padrão, esta opção deve ser utilizada e, provavelmente, também a opção `--with-libraries` correspondente.

Exemplo: `--with-includes=/opt/gnu/include:/usr/sup/include`.

--with-libraries=*DIRETÓRIOS*

*DIRETÓRIOS* é uma lista de diretórios, separados por dois-pontos (:), para procurar pelas bibliotecas. Provavelmente será necessário utilizar esta opção (e a opção --with-includes correspondente), se existirem pacotes instalados em locais que não são o local padrão.

Exemplo: --with-libraries=/opt/gnu/lib:/usr/sup/lib.

--enable-nls[=*IDIOMAS*]

Habilita o Suporte ao Idioma Nativo (Native Language Support - NLS), ou seja, a capacidade de mostrar as mensagens do programa em um idioma diferente do Inglês. *IDIOMAS* é uma lista de códigos de idioma, separados por espaço, a serem suportados, como, por exemplo, --enable-nls='de fr' (A interseção entre a lista especificada e o conjunto de traduções disponíveis atualmente <sup>2</sup> é computada automaticamente). Se esta lista não for especificada, então são instaladas todas as traduções disponíveis. <sup>3</sup>

Para utilizar esta opção é necessária a implementação da API Gettext; veja acima.

--with-pgport=*NÚMERO*

Define *NÚMERO* como o número padrão da porta para o servidor e para os clientes. O valor padrão é 5432 (<http://www.iana.org/assignments/port-numbers>). O número da porta pode ser mudado posteriormente, mas se for especificado aqui então tanto o servidor quanto os clientes terão o mesmo padrão de compilação, o que pode ser bastante conveniente. Geralmente o único bom motivo para escolher um valor diferente do padrão é quando são executados vários servidores PostgreSQL na mesma máquina.

--with-perl

Construir a linguagem PL/Perl do lado servidor.

--with-python

Construir a linguagem PL/Python do lado servidor.

--with-tcl

Construir a linguagem PL/Tcl do lado servidor.

--with-tclconfig=*DIRETÓRIO*

Tcl instala o arquivo `tclConfig.sh`, que contém as informações de configuração necessárias para construir os módulos que interfaceiam com o Tcl. Normalmente este arquivo é encontrado automaticamente em um local bem-conhecido, mas se for desejado utilizar uma versão diferente do Tcl, pode ser especificado o diretório onde esta versão se encontra.

--with-krb4

--with-krb5

Constrói o suporte para autenticação Kerberos. Pode ser utilizado o Kerberos versão 4 ou 5, mas não os dois. Em muitos sistemas operacionais o Kerberos não é instalado no local onde é procurado por padrão (por exemplo, `/usr/include`, `/usr/lib`), portanto é necessário utilizar as opções --with-includes e --with-libraries, além desta opção. Antes de prosseguir, o `configure` verifica os arquivos de cabeçalho e bibliotecas requeridos para ter certeza que a instalação do Kerberos está adequada.

--with-krb-srvnam=*NOME*

O nome do principal do serviço Kerberos. O valor padrão é `postgres`. Provavelmente não existe razão para que seja mudado.

--with-openssl

Construir o suporte às conexões SSL (criptografadas). Requer a instalação do pacote OpenSSL. Antes de prosseguir, o `configure` verifica os arquivos de cabeçalho e bibliotecas requeridos para ter certeza que a instalação do OpenSSL está adequada.

--with-pam

Constrói o suporte a PAM (Módulos de Autenticação Conectáveis/Pluggable Authentication Modules).

--without-readline

Impede a utilização da biblioteca Readline. Desabilita o histórico e a edição de linha de comando no psql e, portanto, não é recomendado.

--with-rendezvous

Constrói o suporte a Rendezvous. Requer suporte a Rendezvous pelo sistema operacional. Recomendado para o Mac OS X.

--disable-spinlocks

Permite a construção ser bem sucedida, mesmo que nesta plataforma o PostgreSQL não tenha suporte de `spinlock` <sup>4</sup> na CPU. A falta de suporte de `spinlock` ocasiona um desempenho pobre; portanto, esta opção deve ser utilizada somente se a construção interromper informando que não há suporte a `spinlock` nesta plataforma. Se esta opção for requerida para construir o PostgreSQL na plataforma sendo utilizada, por favor informe este problema aos desenvolvedores do PostgreSQL.

--enable-thread-safety

Construir as bibliotecas cliente `thread-safe`. <sup>5</sup> Isto permite às `threads` simultâneas nos programas `libpq` e `ECPG` controlarem com segurança seus tratadores de conexão privativos. Esta opção requer suporte a `thread` adequado por parte do sistema operacional.

--without-zlib

Não permite o uso da biblioteca Zlib. Esta opção desabilita o suporte a arquivos comprimidos por parte do `pg_dump`. Esta opção se destina apenas àqueles raros sistemas onde esta biblioteca não está disponível.

--enable-debug

Compila todos os programas e bibliotecas com símbolos de depuração. Isto significa que os programas podem ser executados através do depurador para analisar problemas. Também aumenta consideravelmente o tamanho dos executáveis instalados e, em compiladores não-GCC, geralmente desabilita também a otimização feita pelo compilador, tornando a execução mais lenta. Entretanto, ter os símbolos disponíveis é extremamente útil para lidar com qualquer problema que possa aparecer. Atualmente, esta opção é recomendada para instalações de produção somente se for utilizado o GCC. Mas esta opção deve estar sempre habilitada ao se realizar trabalho de desenvolvimento, ou ao executar uma versão beta.

--enable-cassert

Habilita a verificação das *asserções* <sup>6</sup> no servidor, que testa muitas condições que “não podem acontecer”. É muito útil para fins de desenvolvimento de código, mas os testes tornam a execução um pouco mais lenta. Porém, ter os testes ativos não aumenta a estabilidade do servidor necessariamente! As verificações de asserção não são categorizadas por severidade e, portanto, um erro que pode ser relativamente inofensivo, faz com que o servidor seja reiniciado se for disparada uma falha de asserção. Atualmente esta opção não é recomendada para uso em produção, mas deve ser utilizada ao se realizar trabalho de desenvolvimento ou ao executar uma versão beta.

--enable-depend

Habilita o acompanhamento automático das dependências. Com esta opção, a construção é configurada de forma que todos os arquivos objeto afetados são reconstruídos quando qualquer arquivo de cabeçalho é modificado. É útil ao se realizar trabalho de desenvolvimento, mas é apenas um desperdício de trabalho quando o que se pretende é compilar uma vez e instalar. Atualmente esta opção funciona apenas quando é utilizado o GCC.

Se for preferido utilizar um compilador C diferente do escolhido pelo `configure`, pode ser definida a variável de ambiente `CC` com o programa preferido. Por padrão, o `configure` escolhe o `gcc` se este estiver disponível, senão o padrão da plataforma (geralmente `cc`). De forma semelhante, os sinalizadores de compilação podem ser modificados através da variável `CFLAGS`.

As variáveis de ambiente podem ser especificadas na linha de comando do `configure` como, por exemplo:

```
./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'
```

## 2. Construção:

Para iniciar a construção, deve ser executado

**gmake**

(Lembre-se de usar o make do GNU) A construção pode levar de 5 minutos a meia hora, dependendo da máquina. A última linha exibida deve ser:

```
All of PostgreSQL is successfully made. Ready to install.
```

### 3. Testes de regressão:

Se for desejado testar o servidor recém construído antes de fazer a instalação, podem ser executados os testes de regressão neste ponto. Os testes de regressão são um conjunto de testes que verificam se o PostgreSQL processa na máquina onde os testes são realizados, da maneira como os desenvolvedores esperam que processe. Deve ser executado

```
gmake check
```

(Não funciona sob o usuário root; deve ser executado sob um usuário sem privilégios) O Capítulo 26 contém informações detalhadas sobre como interpretar os resultados dos testes. Posteriormente, este teste poderá ser repetido a qualquer momento executando o mesmo comando.

### 4. Instalação dos arquivos:

**Nota:** Se estiver sendo feita a atualização de um sistema existente, e os novos arquivos vão ser instalados por cima dos antigos, então deve ser feita uma cópia de segurança dos dados e parado o servidor antigo agora, conforme explicado na Seção 14.4 acima.

Para instalar o PostgreSQL, deve ser executado:

```
gmake install
```

Este comando instala os arquivos nos diretórios especificados no passo 1. Deve haver certeza que se possui as permissões apropriadas para escrever nesta área. Normalmente, é necessário executar este passo sob o usuário root. Como alternativa, podem ser previamente criados os diretórios de destino e conseguido que sejam concedidas as permissões apropriadas.

Pode ser utilizado `gmake install-strip`, em vez de `gmake install`, para executar o `strip`<sup>7</sup> nos arquivos executáveis e bibliotecas durante a instalação. Isto reduz um pouco o espaço necessário. Se a construção tiver sido feita com suporte a depuração, fará com que o suporte a depuração seja removido e, portanto, somente deve ser usado quando a depuração não for mais necessária. O `install-strip` tenta executar um bom trabalho de redução de espaço, mas não possui um conhecimento perfeito de como retirar todo byte desnecessário de um arquivo executável, portanto, se for desejado reduzir o espaço ocupado em disco ao máximo, deverá ser realizado um trabalho manual.

A instalação padrão disponibiliza todos os arquivos de cabeçalho necessários para desenvolvimento de aplicativos cliente, assim como desenvolvimento de programas para o lado servidor, como funções personalizadas ou tipos de dado escritos em C (Antes do PostgreSQL 8.0, era necessário executar o comando `a part gmake install-all-headers` para desenvolvimento de programas para o lado servidor, mas este passo foi incorporado à instalação padrão).

**Instalação somente do lado cliente:** Se for desejado instalar apenas os aplicativos cliente e as bibliotecas de interface, então podem ser utilizados os seguintes comandos:

```
gmake -C src/bin install  
gmake -C src/include install  
gmake -C src/interfaces install  
gmake -C doc install
```

**Registro de eventos no Windows:** No Windows, deve ser executado o seguinte comando, após a instalação, para registrar a biblioteca que envia mensagens para o EventLog:

```
regsvr32 pgsql_library_directory/pgevent.dll
```

Esta biblioteca cria as entradas no registro de eventos vistas através do visualizador de eventos.

**Desinstalação:** Para desfazer a instalação deve ser usado o comando `gmake uninstall`. Entretanto, não serão removidos os diretórios criados.

**Limpeza:** Após a instalação, pode ser liberado espaço removendo da árvore de fontes os arquivos construídos, usando o comando `gmake clean`. São mantidos os arquivos produzidos pelo programa `configure` e, portanto, tudo pode ser

reconstruído posteriormente pelo `gmake`. Para retornar a árvore de fontes ao estado em que foi distribuída, deve ser executado `gmake distclean`. Para se fazer construções para várias plataformas a partir da mesma árvore de fontes, deve ser executado `gmake distclean` e feita uma reconfiguração para cada construção (Como alternativa, pode ser usada uma árvore de construção separada para cada plataforma, de forma que a árvore de fontes permaneça sem modificações).

Se for feita uma construção e depois descoberto que as opções do `configure` estavam erradas, ou se for modificado algo que o `configure` investiga (por exemplo, atualizações de software), então é uma boa idéia executar `gmake distclean` antes de reconfigurar e reconstruir. Se isto não for feito, as modificações feitas nas escolhas de configuração podem não se propagar para todos os lugares onde há necessidade.

## 14.6. Configurações de pós-instalação

### 14.6.1. Bibliotecas compartilhadas

Em alguns sistemas que possuem bibliotecas compartilhadas (o que a maioria tem) é necessário informar ao sistema como encontrar as bibliotecas compartilhadas recém instaladas. Os sistemas onde *não* há necessidade incluem BSD/OS, FreeBSD, HP-UX, IRIX, Linux, NetBSD, OpenBSD, Tru64 UNIX (antigo Digital UNIX) e Solaris.

O método para definir o caminho de procura de biblioteca compartilhada varia entre plataformas, mas o método mais amplamente utilizado é definir a variável de ambiente `LD_LIBRARY_PATH` da seguinte forma: Nos interpretadores de comando Bourne (`sh`, `ksh`, `bash`, `zsh`)

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

ou em `csh` e `tcsh`

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

Deve ser substituído `/usr/local/pgsql/lib` pelo que foi definido para `--libdir` no passo 1. Estes comandos devem ser colocados no arquivo de inicialização do interpretador de comandos, como `/etc/profile` ou `~/.bash_profile`. Podem ser encontradas algumas boas informações sobre problemas associados a este método em <http://www.visi.com/~barr/ldpath.html>.

Em alguns sistemas pode ser preferível definir a variável de ambiente `LD_RUN_PATH` *antes* da construção.

No Cygwin o diretório da biblioteca deve ser colocado no `PATH`, ou mover os arquivos `.dll` para o diretório `bin`.

Em caso de dúvida, deve ser consultada as páginas do manual do sistema operacional (talvez `ld.so` <sup>8</sup> ou `rld` <sup>9</sup>). Se posteriormente for recebida uma mensagem como

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or directory
```

então este passo era necessário. Simplesmente cuide de fazê-lo.

Se estiver sendo utilizado o BSD/OS, Linux ou SunOS 4, e se possuir acesso como `root` pode ser executado

```
/sbin/ldconfig /usr/local/pgsql/lib
```

(ou um diretório equivalente) após a instalação, para fazer com que o ligador em tempo de execução encontre as bibliotecas compartilhadas mais rapidamente. Para obter informações adicionais sobre `ldconfig` <sup>10</sup> deve ser consultada a página do manual. No FreeBSD, NetBSD e OpenBSD o comando é

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

em vez do mostrado anteriormente. Desconhecemos comando equivalente em outros sistemas.

### 14.6.2. Variáveis de Ambiente

Se a instalação for feita em `/usr/local/pgsql`, ou em algum outro local onde não seja feita por padrão a procura por programas, deve ser adicionado à variável de ambiente `PATH` o diretório `/usr/local/pgsql/bin` (ou o que foi definido para `--bindir` no passo 1). A rigor isto não é necessário, mas torna a utilização do PostgreSQL muito mais confortável.

Para fazer isto, deve ser adicionado ao arquivo de inicialização do interpretador de comandos, como `~/.bash_profile` (ou `/etc/profile`, se for para todos os usuários):

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

Se estiver sendo utilizado `csch` ou `tcsh`, então deve ser utilizado o comando:

```
set path = ( /usr/local/pgsql/bin $path )
```

Para permitir o sistema encontrar a documentação `man`, é necessário adicionar linhas como as mostradas abaixo ao arquivo de inicialização do interpretador de comandos, a menos que seja instalado em um local procurado por padrão.

```
MANPATH=/usr/local/pgsql/man:$MANPATH
export MANPATH
```

As variáveis de ambiente `PGHOST` e `PGPORT` especificam, para os aplicativos cliente, o hospedeiro e a porta do servidor de banco de dados, prevalecendo sobre os padrões de compilação. Se forem executadas aplicativos cliente remotamente, então é conveniente que todo usuário com intenção de utilizar o banco de dados defina `PGHOST`. Entretanto, não é obrigatório: as definições podem ser comunicadas através das opções de linha de comando para a maioria dos programas cliente.

## 14.7. Plataformas suportadas

A comunidade de desenvolvedores verificou o funcionamento do PostgreSQL nas plataformas listadas abaixo. Uma plataforma suportada significa, geralmente, que o PostgreSQL pode ser construído e instalado de acordo com estas instruções e que passa nos testes de regressão. As entradas “Fazenda de construção” (Build farm) se referem às construções relatadas em PostgreSQL Build Farm (<http://www.pgbuildfarm.org/>). As entradas de plataforma que mostram versões antigas do PostgreSQL são as que não receberam testes explícitos até o momento da liberação da versão 8.0, mas que ainda se espera que funcionem.

**Nota:** Em caso de problemas de instalação em uma plataforma suportada, por favor escreva para `<pgsql-bugs@postgresql.org>` ou para `<pgsql-ports@postgresql.org>`, e não para as pessoas que aqui aparecem.

S.O.	Processador	Versão	Relatado	Comentários
AIX	PowerPC	8.0.0	Travis P ( <code>&lt;twp@castle.fastmail.fm&gt;</code> ), 2004-12-12	consulte também <code>doc/FAQ_AIX</code>
AIX	RS6000	8.0.0	Hans-Jürgen Schönig ( <code>&lt;hs@cybertec.at&gt;</code> ), 2004-12-06	consulte também <code>doc/FAQ_AIX</code>
BSD/OS	x86	8.0.0	Bruce Momjian ( <code>&lt;pgman@candle.pha.pa.us&gt;</code> ), 2004-12-07	4.3.1
Debian GNU/Linux	Alpha	7.4	Noël Köthe ( <code>&lt;noel@debian.org&gt;</code> ), 2003-10-25	
Debian GNU/Linux	AMD64	8.0.0	Fazenda de construção panda, instantâneo 2004-12-06 01:20:02	sid, kernel 2.6
Debian GNU/Linux	ARM	8.0.0	Jim Buttafuoco ( <code>&lt;jim@contactbda.com&gt;</code> ), 2005-01-06	
Debian GNU/Linux	IA64	7.4	Noël Köthe ( <code>&lt;noel@debian.org&gt;</code> ), 2003-10-25	
Debian GNU/Linux	m68k	8.0.0	Noël Köthe ( <code>&lt;noel@debian.org&gt;</code> ), 2004-12-09	sid
Debian GNU/Linux	MIPS	8.0.0	Fazenda de construção lionfish, instantâneo 2004-12-06 11:00:08	3.1 (sarge), kernel 2.4
Debian	PA-RISC	8.0.0	Noël Köthe ( <code>&lt;noel@debian.org&gt;</code> ), 2004-12-	sid



S.O.	Processador	Versão	Relatado	Comentários
GNU/Linux			07	
Debian GNU/Linux	PowerPC	8.0.0	Noël Köthe (<noel@debian.org>), 2004-12-15	sid
Debian GNU/Linux	S/390	7.4	Noël Köthe (<noel@debian.org>), 2003-10-25	
Debian GNU/Linux	Sparc	8.0.0	Noël Köthe (<noel@debian.org>), 2004-12-09	sid, 32-bit
Debian GNU/Linux	x86	8.0.0	Peter Eisentraut (<peter_e@gmx.net>), 2004-12-06	3.1 (sarge), kernel 2.6
Fedora	AMD64	8.0.0	John Gray (<jgray@azuli.co.uk>), 2004-12-12	FC3
Fedora	x86	8.0.0	Fazenda de construção dog, instantâneo 2004-12-06 02:06:01	FC1
FreeBSD	Alpha	7.4	Peter Eisentraut (<peter_e@gmx.net>), 2003-10-25	4.8
FreeBSD	x86	8.0.0	Fazenda de construção cockatoo, instantâneo 2004-12-06 14:10:01 (4.10); Marc Fournier (<scrappy@postgresql.org>), 2004-12-07 (5.3)	
Gentoo Linux	AMD64	8.0.0	Jani Averbach (<jaa@jaa.iki.fi>), 2005-01-13	
Gentoo Linux	x86	8.0.0	Paul Bort (<pbort@tmwsystems.com>), 2004-12-07	
HP-UX	IA64	8.0.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2005-01-06	11.23, gcc and cc; consulte também doc/FAQ_HPUX
HP-UX	PA-RISC	8.0.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2005-01-06	10.20 and 11.11, gcc and cc; consulte também doc/FAQ_HPUX
IRIX	MIPS	7.4	Robert E. Brucoleri (<bruc@stone.congenomics.com>), 2003-11-12	6.5.20, somente cc
Mac OS X	PowerPC	8.0.0	Andrew Rawnsley (<ronz@ravensfield.com>), 2004-12-07	10.3.5
Mandrakelinux	x86	8.0.0	Fazenda de construção shrew, instantâneo 2004-12-06 02:02:01	10.0
NetBSD	arm32	7.4	Patrick Welche (<prlw1@newn.cam.ac.uk>), 2003-11-12	1.6ZE/acorn32
NetBSD	m68k	8.0.0	Rémi Zara (<remi_zara@mac.com>), 2004-12-14	2.0
NetBSD	Sparc	7.4.1	Peter Eisentraut (<peter_e@gmx.net>), 2003-11-26	1.6.1, 32-bit

S.O.	Processador	Versão	Relatado	Comentários
NetBSD	x86	8.0.0	Fazenda de construção canary, instantâneo 2004-12-06 03:30:00	1.6
OpenBSD	Sparc	8.0.0	Chris Mair (<list@1006.org>), 2005-01-10	3.3
OpenBSD	Sparc64	8.0.0	Fazenda de construção spoonbill, instantâneo 2005-01-06 00:50:05	3.6
OpenBSD	x86	8.0.0	Fazenda de construção emu, instantâneo 2004-12-06 11:35:03	3.6
Red Hat Linux	AMD64	8.0.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2004-12-07	RHEL 3AS
Red Hat Linux	IA64	8.0.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2004-12-07	RHEL 3AS
Red Hat Linux	PowerPC	8.0.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2004-12-07	RHEL 3AS
Red Hat Linux	PowerPC 64	8.0.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2004-12-07	RHEL 3AS
Red Hat Linux	S/390	8.0.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2004-12-07	RHEL 3AS
Red Hat Linux	S/390x	8.0.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2004-12-07	RHEL 3AS
Red Hat Linux	x86	8.0.0	Tom Lane (<tgl@sss.pgh.pa.us>), 2004-12-07	RHEL 3AS
Solaris	Sparc	8.0.0	Kenneth Marshall (<ktm@is.rice.edu>), 2004-12-07	Solaris 8; consulte também doc/FAQ_Solaris
Solaris	x86	8.0.0	Fazenda de construção kudu, instantâneo 2004-12-10 02:30:04 (cc); dragonfly, snapshot 2004-12-09 04:30:00 (gcc)	Solaris 9; consulte também doc/FAQ_Solaris
SUSE Linux	AMD64	8.0.0	Reinhard Max (<max@suse.de>), 2005-01-03	9.0, 9.1, 9.2, SLES 9
SUSE Linux	IA64	8.0.0	Reinhard Max (<max@suse.de>), 2005-01-03	SLES 9
SUSE Linux	PowerPC	8.0.0	Reinhard Max (<max@suse.de>), 2005-01-03	SLES 9
SUSE Linux	PowerPC 64	8.0.0	Reinhard Max (<max@suse.de>), 2005-01-03	SLES 9
SUSE Linux	S/390	8.0.0	Reinhard Max (<max@suse.de>), 2005-01-03	SLES 9
SUSE Linux	S/390x	8.0.0	Reinhard Max (<max@suse.de>), 2005-01-03	SLES 9
SUSE Linux	x86	8.0.0	Reinhard Max (<max@suse.de>), 2005-01-03	9.0, 9.1, 9.2, SLES 9
Tru64 UNIX	Alpha	8.0.0	Honda Shigehiro (<fwif0083@mb.infoweb.ne.jp>), 2005-01-07	5.0
UnixWare	x86	8.0.0	Peter Eisentraut (<peter_e@gmx.net>), 2004-12-14	cc, 7.1.4; consulte também doc/FAQ_SCO
Windows	x86	8.0.0	Dave Page (<dpage@vale-housing.co.uk>), 2004-12-07	XP Pro; consulte doc/FAQ_MINGW

S.O.	Processador	Versão	Relatado	Comentários
Windows with Cygwin	x86	8.0.0	Fazenda de construção gibbon, instantâneo 2004-12-11 01:33:01	consulte doc/FAQ_CYGWIN

**Plataformas não suportadas:** Nas seguintes plataformas ou se sabe que não funciona, ou que funcionava em uma versão anterior muito remota. São incluídas para que se saiba que estas plataformas *podem* ser suportadas se for dada alguma atenção.

S.O.	Processador	Versão	Relatado	Comentários
BeOS	x86	7.2	Cyril Velter (<cyril.velter@libertysurf.fr>), 2001-11-29	necessita atualização para o código de semáforo
Linux	PlayStation 2	8.0.0	Chris Mair (<list@1006.org>), 2005-01-09	requer --disable-spinlocks (funciona, mas lentamente)
NetBSD	Alpha	7.2	Thomas Thai (<tom@minnesota.com>), 2001-11-20	1.5W
NetBSD	MIPS	7.2.1	Warwick Hunter (<whunter@agile.tv>), 2002-06-13	1.5.3
NetBSD	PowerPC	7.2	Bill Studenmund (<wrstuden@netbsd.org>), 2001-11-28	1.5
NetBSD	VAX	7.1	Tom I. Helbekkmo (<tih@kpnqwest.no>), 2001-03-30	1.5
QNX 4 RTOS	x86	7.2	Bernd Tegge (<tegge@repas-aeg.de>), 2001-12-10	necessita atualização para o código de semáforo; consulte também doc/FAQ_QNX4
QNX RTOS v6	x86	7.2	Igor Kovalenko (<Igor.Kovalenko@motorola.com>), 2001-11-20	correções disponíveis em arquivos, mas muito tarde para a 7.2
SCO OpenServer	x86	7.3.1	Shibashish Satpathy (<shib@postmark.net>), 2002-12-11	5.0.4, gcc; consulte também doc/FAQ_SCO
SunOS 4	Sparc	7.2	Tatsuo Ishii (<t-ishii@sra.co.jp>), 2001-12-04	

## 14.8. Instalação no Fedora Core 3

**Nota:** Seção escrita pelo tradutor, não fazendo parte do manual original.

Esta seção mostra como instalar o PostgreSQL 8.0 no Fedora Core 3, a partir do arquivo postgresql-8.0.0.tar.gz baixado no diretório /download, através do usuário root.

```
# cd /tmp
# tar xzvf /download/postgresql-8.0.0.tar.gz
# cd postgresql-8.0.0/
# ./configure --enable-nls
# make
All of PostgreSQL successfully made. Ready to install.
# make install
PostgreSQL installation complete.
```

```
# # Criar o usuário postgres
# adduser postgres
# # Criar o diretório de dados
# mkdir /usr/local/pgsql/data
# # Tornar o usuário postgres o dono do diretório de dados
# chown postgres:postgres /usr/local/pgsql/data
# # Se tornar o usuário postgres
# su - postgres
# # Inicializar a área de dados
$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
WARNING: enabling "trust" authentication for local connections
You can change this by editing pg_hba.conf or using the -A option the
next time you run initdb.
Success. You can now start the database server using:
    /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
or
    /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start
# # Ativar o servidor PostgreSQL
$ /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data >logfile 2>&1 &
# # Criar o usuário teste com a senha teste
$ /usr/local/pgsql/bin/createuser teste --pwprompt
Forneça a senha do novo usuário:
Forneça novamente:
Será permitido ao novo usuário criar bancos de dados? (s/n) n
Será permitido ao novo usuário criar outros usuários? (s/n) n
CREATE USER
# # Criar o banco de dados teste tendo por dono o usuário teste
$ /usr/local/pgsql/bin/createdb teste --owner teste
CREATE DATABASE
# # Usar o psql para acessar o banco de dados teste
$ /usr/local/pgsql/bin/psql -U teste teste
Bem-vindo ao psql 8.0.0, o terminal interativo do PostgreSQL.
```

```
Digite: \copyright para mostrar a licença da distribuição
        \h para ajuda nos comandos SQL
        \? para ajuda nos comandos do psql
        \g ou finalizar com ponto-e-vírgula para executar o comando
        \q para sair
```

```
teste=> \l
```

```
      Lista dos bancos de dados
  Nome      |  Dono   | Codificação
-----+-----+-----
template0 | postgres | LATIN1
templatel  | postgres | LATIN1
teste      | teste    | LATIN1
(3 linhas)
```

```
teste=> \q
# # Definir o caminho de procura das bibliotecas
# LD_LIBRARY_PATH=/usr/local/pgsql/lib/
# export LD_LIBRARY_PATH
# # Definir o caminho de procura dos executáveis
# PATH=$PATH:/usr/local/pgsql/bin/:
# export PATH
```

**Nota:** Para ver as páginas do manual usando o comando `man` deve ser adicionada a linha `MANPATH /usr/local/pgsql/man/` ao arquivo `/etc/man.config`.

Abaixo está mostrado o script `/etc/init.d/postgresql` para efetuar as operações de ativar, parar, mostrar o status, reativar e recarregar o servidor PostgreSQL.

```

#!/bin/sh
PGDATA=/usr/local/pgsql/data
PGBIN=/usr/local/pgsql/bin
PGLOG=/dev/null
start(){
    echo "Ativando o serviço PostgreSQL: "
    su -l postgres -c "$PGBIN/pg_ctl -D $PGDATA start >> $PGLOG"
}

stop(){
    echo -n $"Parando o serviço PostgreSQL: "
    su -l postgres -c "$PGBIN/pg_ctl -D $PGDATA stop"
}

status(){
    echo "Status do serviço PostgreSQL: "
    su -l postgres -c "$PGBIN/pg_ctl -D $PGDATA status"
}

restart(){
    echo "Reativando o serviço PostgreSQL: "
    su -l postgres -c "$PGBIN/pg_ctl -D $PGDATA restart"
}

reload(){
    echo "Recarregando o serviço PostgreSQL: "
    su -l postgres -c "$PGBIN/pg_ctl -D $PGDATA reload"
}
case "$1" in
    start)
        start
        ;;
    stop)
        stop
        ;;
    status)
        status
        ;;
    restart)
        restart
        ;;
    reload|force-reload)
        reload
        ;;
    *)
        echo $"Usage: $0 {start|stop|status|restart|reload|force-reload}"
        exit 1
esac

exit 0

```

## Notas

1. No Fedora Core 3 o `gmake` é um vínculo simbólico para o `make` (`/usr/bin/gmake -> make`). (N. do T.)
2. Traduções disponíveis atualmente — `AVAIL_LANGUAGES` := `af cs de es fr hr hu it ko nb pt_BR ro ru sk sl sv tr zh_CN zh_TW` (N. do T.)
3. A configuração padrão é sem suporte ao idioma nativo, portanto, se este parâmetro não for especificado, não será possível ver as mensagens em português (N. do T.).
4. `spinlock` — em engenharia de software é um bloqueio onde cada fluxo de execução (`thread`) simplesmente aguarda em um laço (`gira/spins`) verificando repetidamente até que o bloqueio fique disponível. Free Online Dictionary and Thesaurus (<http://encyclopedia.thefreedictionary.com/Spin%20lock>) (N. do T.)
5. `thread-safe` — (programação) A descrição de um código que é reentrante ou protegido em várias execuções simultâneas por alguma forma de exclusão mútua. FOLDOC - Free On-Line Dictionary of Computing (<http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=thread-safe>) (N. do T.)
6. `assertão` — do Lat. `assertione` — proposição que se apresenta como verdadeira. PRIBERAM - Língua Portuguesa On-Line (<http://www.priberam.pt/dlpo/dlpo.aspx>). (N. do T.)
7. `strip` — (comando do Unix) O comando `strip` remove a tabela de símbolos, informação de depuração e informação de número de linha de arquivos objeto ELF. Após ter sido realizado o processo de `strip`, não haverá nenhum acesso de depuração simbólica para este arquivo; normalmente este comando só é executado em módulos de produção que já foram depurados e testados. Man Page (<http://www.cs.princeton.edu/cgi-bin/man2html?strip:1>) (N. do T.)
8. `ld.so` — ligador em tempo de execução (`run-time linker`) para objetos dinâmicos. Página do Manual (<http://mirrors.ccs.neu.edu/cgi-bin/unixhelp/man-cgi?ld.so.1+1>) (N. do T.)
9. `rld` — ligador em tempo de execução (`run-time linker`). Página do Manual (<http://www.mcsr.olemiss.edu/cgi-bin/man-cgi?rld+1>) (N. do T.)
10. `ldconfig` — determina os vínculos de ligação em tempo de execução. Página do Manual (<http://www.rt.com/man/ldconfig.8.html>) (N. do T.)

## Capítulo 15. Instalação apenas do cliente no Windows

Embora uma instalação completa do PostgreSQL para o Windows somente possa ser feita utilizando o MinGW ou o Cygwin, a biblioteca cliente C (libpq) e o terminal interativo (psql) podem ser compilados utilizando outros conjuntos de ferramentas do Windows. São incluídos na distribuição do código fonte arquivos de construção escritos para o Microsoft Visual C++ e para o Borland C++. Para outras configurações deve ser possível compilar as bibliotecas manualmente.

**Dica:** A utilização de MinGW ou de Cygwin são as formas preferidas. Caso seja utilizado outro conjunto de ferramentas deve ser visto o Capítulo 14.

Para construir tudo que é possível no Windows utilizando o Microsoft Visual C++, o diretório `src` deve ser tornado o diretório corrente e executado o comando:

```
nmake /f win32.mak
```

Este comando pressupõe a existência do Visual C++ no caminho de procura.

Para construir tudo que é possível utilizando o Borland C++, o diretório `src` deve ser tornado o diretório corrente e executado o comando:

```
make -DCFG=Release /f bcc32.mak
```

São construídos os seguintes arquivos:

```
interfaces\libpq\Release\libpq.dll
```

A biblioteca cliente vinculável dinamicamente

```
interfaces\libpq\Release\libpqdll.lib
```

A biblioteca de importação para vincular os programas à biblioteca `libpq.dll`

```
interfaces\libpq\Release\libpq.lib
```

A versão estática da biblioteca cliente

```
bin\psql\Release\psql.exe
```

O terminal interativo do PostgreSQL

A biblioteca `libpq.dll` é o único arquivo que realmente precisa ser instalado. Na maioria dos casos este arquivo deve ser colocado no diretório `WINNT\SYSTEM32` (ou no diretório `WINDOWS\SYSTEM` no Windows 95/98/ME). Se este arquivo for instalado através do programa de instalação, deve ser instalado com verificação de versão utilizando o recurso `VERSIONINFO` incluído no arquivo, para garantir que caso haja uma versão mais nova da biblioteca, esta não será sobrescrita.

Se for planejado fazer desenvolvimento nesta máquina utilizando a biblioteca `libpq`, então é necessário adicionar ao caminho de inclusão, na configuração do compilador, os subdiretórios `src\include` e `src\interfaces\libpq` da árvore do código fonte.

Para utilizar a biblioteca, deve ser adicionado o arquivo `libpqdll.lib` ao projeto (No Visual C++ deve-se, simplesmente, dar um clique com o botão direito do mouse no projeto e escolher adicioná-lo).

# Capítulo 16. Ambiente do servidor em tempo de execução

Este capítulo mostra como configurar e executar o servidor de banco de dados, e as interações do servidor com o sistema operacional.

## 16.1. A conta de usuário do PostgreSQL

Como todos os outros processos servidor (*daemon*)<sup>1</sup> que podem ser acessados pelo mundo exterior, é aconselhável executar o PostgreSQL sob uma conta de usuário em separado. Esta conta de usuário somente deve possuir os dados gerenciados pelo servidor, não devendo ser compartilhada por outros processos servidor; por exemplo, é uma má idéia utilizar o usuário *nobody*. Não se aconselha instalar executáveis cujo dono seja este usuário, porque sistemas comprometidos poderiam modificar seus próprios binários.

Para adicionar uma conta de usuário Unix ao sistema, deve ser procurado o comando *useradd* ou *adduser*. Normalmente é utilizado o nome de usuário *postgres*, que também é o nome assumido nesta documentação, mas se for preferido pode ser utilizado outro nome.

## 16.2. Criação do agrupamento de bancos de dados

Antes de ser possível fazer qualquer coisa, deve ser inicializada a área em disco para armazenamento dos bancos de dados. Isto é chamado de criação do *agrupamento de bancos de dados* (*database cluster*); em vez desse nome, o padrão SQL utiliza o termo agrupamento de catálogos (*catalog cluster*). Um agrupamento de bancos de dados é uma coleção de bancos de dados gerenciada por uma única instância de um servidor de banco de dados em execução. Após ser criado, o agrupamento de bancos de dados contém um banco de dados chamado *template1*. Como o nome sugere (em inglês), este banco de dados é utilizado como modelo para os próximos bancos de dados a serem criados; não deve ser utilizado para trabalho (Para obter informações sobre a como criar bancos de dados no agrupamento deve ser visto o Capítulo 18).

Em termos de sistema de arquivos, um agrupamento de bancos de dados é um único diretório sob o qual todos os dados são armazenados. Este diretório é chamado de *diretório de dados* ou *área de dados*. A escolha do local onde os dados são armazenados depende inteiramente de quem faz a instalação. Não existe nenhum padrão, embora sejam usuais locais como */usr/local/pgsql/data* e */var/lib/pgsql/data*. Para inicializar um agrupamento de bancos de dados é utilizado o utilitário *initdb*, que é instalado junto com o PostgreSQL. O local no sistema de arquivos escolhido para o agrupamento de bancos de dados é indicado pela opção *-D* como, por exemplo:

```
$ initdb -D /usr/local/pgsql/data
```

Deve ser observado que este comando deve ser executado enquanto conectado na conta de usuário do PostgreSQL, conforme descrito na seção anterior.

**Dica:** Pode ser definida a variável de ambiente *PGDATA* como alternativa ao uso da opção *-D*.

O utilitário *initdb* tenta criar o diretório especificado, caso não exista. É provável que não tenha permissão para criar o diretório (se for seguido o conselho de criar um usuário sem privilégios). Neste caso, o diretório deve ser criado utilizando a conta de usuário *root*, e mudado o dono do diretório para o usuário do PostgreSQL. Abaixo está mostrado como deve ser feito:

```
root# mkdir /usr/local/pgsql/data
root# chown postgres /usr/local/pgsql/data
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

O utilitário *initdb* se recusa a executar quando o diretório de dados parece que já foi inicializado.

Como o diretório de dados contém todos os dados armazenados no banco de dados, é essencial que esteja seguro contra acessos não autorizados. Portanto, o utilitário *initdb* revoga as permissões de acesso de todos, menos do usuário do PostgreSQL.

Entretanto, embora o conteúdo do diretório esteja seguro, a configuração de autenticação de cliente padrão permite qualquer usuário local se conectar ao banco de dados, e até mesmo se tornar o superusuário do banco de dados. Caso não se



confie nos outros usuários locais, recomenda-se utilizar a opção `-W`, `--pwprompt` ou `--pwfile` do `initdb`, para atribuir uma senha para o superusuário do banco de dados. Também pode ser especificado `-A md5` ou `-A password` para que não seja utilizado o modo de autenticação padrão `trust`; também pode ser modificado o arquivo `pg_hba.conf` gerado após a execução do utilitário `initdb`, antes de iniciar o servidor pela primeira vez (Outras abordagens razoáveis incluem a utilização da autenticação `ident` e permissões do sistema de arquivos para restringir as conexões. Para obter informações adicionais deve ser consultado o Capítulo 19).

O utilitário `initdb` também inicializa o idioma padrão para o agrupamento de bancos de dados. Normalmente, simplesmente pega as definições de idioma do ambiente e aplica ao banco de dados inicializado. É possível especificar um idioma diferente para o banco de dados; informações adicionais sobre este assunto podem ser encontradas na Seção 20.1. A ordem de classificação utilizada por um determinado agrupamento de bancos de dados é definida pelo utilitário `initdb`, e não pode ser mudada posteriormente, a não ser fazendo cópia de segurança de todos os dados, executando novamente o `initdb` e recarregando os dados. Também existe impacto no desempenho quando se utiliza um idioma diferente de `C` ou `POSIX`. Portanto, é importante fazer a escolha correta da primeira vez.

O utilitário `initdb` também define a codificação de caracteres padrão para o agrupamento de bancos de dados. Normalmente deve ser escolhida uma codificação que corresponda à definição do idioma. Para obter detalhes deve ser consultada a Seção 20.2.

### 16.3. Inicialização do servidor de banco de dados

Antes que os usuários possam acessar o banco de dados, o servidor de banco de dados deve ser colocado em execução. O programa servidor de banco de dados chama-se `postmaster`. O `postmaster` precisa saber onde se encontram os dados a serem utilizados. Isto é feito através da opção `-D`. Portanto, a maneira mais simples de colocar o servidor em execução é usando

```
$ postmaster -D /usr/local/pgsql/data
```

que deixa o servidor processando em primeiro plano. Deve ser executado conectado à conta de usuário do PostgreSQL. Sem a opção `-D` o servidor tenta utilizar o diretório de dados especificado na variável de ambiente `PGDATA`. Se esta variável também não existir, então a inicialização falha.

Normalmente é melhor executar o `postmaster` em segundo plano, e neste caso a sintaxe usual para o interpretador de comandos é:

```
$ postmaster -D /usr/local/pgsql/data >arquivo_de_log 2>&1 &
```

É importante armazenar as saídas `stdout` e `stderr` do servidor em algum lugar, conforme mostrado acima. Isto é útil para fins de auditoria e para diagnosticar problemas (A Seção 21.3 contém uma explicação mais detalhada sobre o manuseio do arquivo de `log`).

O `postmaster` também aceita várias outras opções de linha de comando. Para obter informações adicionais deve ser consultada a página de referência do `postmaster` e a Seção 16.4 abaixo.

A sintaxe do interpretador de comandos pode se tornar entediante em pouco tempo. Por isso é fornecido o programa `pg_ctl` para simplificar algumas tarefas. Por exemplo,

```
pg_ctl start -l arquivo_de_log
```

inicializa o servidor em segundo plano, e envia a saída para o arquivo de `log` especificado. A opção `-D` possui o mesmo significado para este programa que no `postmaster`. O programa `pg_ctl` também pode parar o servidor.

Normalmente se quer que o servidor de banco de dados seja inicializado quando o computador é ligado. Os scripts de auto-inicialização são específicos de cada sistema operacional. Existem alguns scripts distribuídos junto com o PostgreSQL no diretório `contrib/start-scripts`. Para instalar estes scripts é necessário o privilégio de `root`.

Sistemas diferentes possuem convenções diferentes para inicializar processos durante a inicialização do sistema operacional. Muitos sistemas possuem o arquivo `/etc/rc.local` ou `/etc/rc.d/rc.local`. Outros utilizam diretórios `rc.d`. Seja da maneira que for, o servidor deve executar sob a conta de usuário do PostgreSQL, e não sob `root` ou qualquer outro usuário. Portanto, os comandos provavelmente devem ser construídos utilizando `su -c '...' postgres`. Por exemplo:

```
su -c 'pg_ctl start -D /usr/local/pgsql/data -l arquivo_de_log' postgres
```

Abaixo são mostradas algumas sugestões mais específicas para vários sistemas operacionais (Os valores genéricos devem ser sempre substituídos pelo diretório de instalação e nome de usuário apropriados para cada caso).

- No FreeBSD deve ser visto o arquivo `contrib/start-scripts/freebsd` na distribuição do código fonte do PostgreSQL.
- No OpenBSD devem ser adicionadas as seguintes linhas ao arquivo `/etc/rc.local`:

```
if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/postmaster ]; then
    su - -c '/usr/local/pgsql/bin/pg_ctl start -l /var/postgresql/arquivo_de_log -s'
    postgres
    echo -n ' postgresql'
fi
```

- Nos sistemas Linux deve ser adicionado

```
/usr/local/pgsql/bin/pg_ctl start -l arquivo_de_log -D /usr/local/pgsql/data
```

ao arquivo `/etc/rc.d/rc.local`, ou ser visto o arquivo `contrib/start-scripts/linux` na distribuição do código fonte do PostgreSQL.

- No NetBSD deve ser utilizado o script de inicialização do FreeBSD ou do Linux, conforme se preferir.
- No Solaris deve ser criado um arquivo chamado `/etc/init.d/postgresql` contendo a seguinte linha:

```
su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l arquivo_de_log -D /usr/local/pgsql/data"
```

Depois deve ser criado um vínculo simbólico para este arquivo em `/etc/rc3.d` como `S99postgresql`.

Enquanto o `postmaster` está executando, seu identificador de processo (PID) fica armazenado no arquivo `postmaster.pid` no diretório de dados. Este arquivo é utilizado para impedir que mais de um processo `postmaster` execute usando o mesmo diretório de dados. Também pode ser utilizado para parar o processo `postmaster`.

### 16.3.1. Falhas na inicialização do servidor

Existem vários motivos triviais pelos quais a inicialização do servidor pode não ser bem-sucedida. Deve ser visto o arquivo de `log` do servidor, ou deve ser feita a inicialização manual do servidor (sem redirecionar a saída padrão ou o erro padrão) para ver as mensagens de erro mostradas. Abaixo são explicadas detalhadamente algumas das mensagens de erro mais comuns.

```
LOG:   could not bind IPv4 socket: Address already in use
HINT:  Is another postmaster already running on port 5432?
       If not, wait a few seconds and retry.
FATAL: could not create TCP/IP listen socket
```

-- Tradução da mensagem

```
LOG:   não foi possível vincular soquete IPv4: Endereço já sendo usado
DICA:  Há outro postmaster usando a porta 5432?
       Se não houver, aguarde uns poucos segundos e tente novamente.
FATAL: não foi possível criar soquete TCP/IP de atendimento
```

Normalmente esta mensagem significa o que sugere: tentou-se inicializar um `postmaster` na mesma porta que outro já estava usando. Entretanto, se o núcleo da mensagem de erro não for *Endereço já sendo usado* (*Address already in use*), ou alguma variação desta, pode estar ocorrendo um problema diferente. Por exemplo, tentar inicializar o `postmaster` em um número de porta reservado pode levar a algo como:

```
$ postmaster -p 666
```

```
LOG:   could not bind IPv4 socket: Permission denied
HINT:  Is another postmaster already running on port 666?
       If not, wait a few seconds and retry.
FATAL: could not create TCP/IP listen socket
```

-- Tradução da mensagem

```
LOG: não foi possível vincular soquete IPv4: Permissão negada
DICA: Há outro postmaster usando a porta 666?
      Se não houver, aguarde uns poucos segundos e tente novamente.
FATAL: não foi possível criar soquete TCP/IP de atendimento
```

Uma mensagem como

```
FATAL: could not create shared memory segment: Invalid argument
DETAIL: Failed system call was shmget(key=5440001, size=4011376640, 03600).
```

-- Tradução da mensagem

```
FATAL: não foi possível criar o segmento de memória compartilhada: Argumento inválido
DETALHE: A chamada de sistema que falhou foi shmget(chave=5440001, tamanho=4011376640, 03600).
```

provavelmente significa que o limite para o tamanho da memória compartilhada do núcleo (kernel) é menor que a área de trabalho que o PostgreSQL está tentando criar (4011376640 bytes neste exemplo). Pode significar, também, que o núcleo não está configurado para dar suporte a memória compartilhada no estilo System-V. Como recurso temporário pode-se tentar inicializar o servidor com um número de buffers menor que o número normal (sinalizador -B). Mais tarde poderá ser necessário reconfigurar o núcleo para aumentar o tamanho de memória compartilhada permitido. Esta mensagem também pode ser vista quando se tenta inicializar vários servidores na mesma máquina, quando o espaço total requisitado excede o limite do núcleo.

Um erro como

```
FATAL: could not create semaphores: No space left on device
DETAIL: Failed system call was semget(5440126, 17, 03600).
```

-- Tradução da mensagem

```
FATAL: não foi possível criar semáforos: Não há espaço suficiente na unidade
DETALHE: A chamada de sistema que falhou foi semget(5440126, 17, 03600).
```

*não* significa que o espaço em disco está esgotado. Significa que o limite do núcleo para o número de semáforos System V é menor que o número de semáforos que o PostgreSQL deseja criar. Como acima, este problema pode ser evitado inicializando o servidor com um número reduzido de conexões permitidas (sinalizador -N), e posteriormente aumentando o limite do núcleo.

Caso ocorra a mensagem de erro “chamada de sistema ilegal”, é provável que o núcleo não tenha suporte para memória compartilhada ou semáforos. Neste caso, a única opção é reconfigurar o núcleo para habilitar estas funcionalidades.

Na Seção 16.5.1 são mostrados detalhes sobre a configuração das facilidades de IPC do System V.

### 16.3.2. Problemas na conexão do cliente

Embora as condições de erro possíveis no lado cliente sejam bastante variadas e dependentes do aplicativo, algumas delas podem estar diretamente relacionadas com a maneira como o servidor é inicializado. Outras condições diferentes das mostradas abaixo devem estar documentadas no próprio aplicativo cliente.

```
psql: could not connect to server: Connection refused
       Is the server running on host "server.joel.com" and accepting
       TCP/IP connections on port 5432?
```

-- Tradução da mensagem

```
psql: não foi possível conectar com o servidor. Conexão recusada
       O servidor estão executando no hospedeiro "server.joel.com"
       e aceitando conexões TCP/IP na porta 5432?
```

Esta é uma falha genérica dizendo “Eu não pude encontrar o servidor para me comunicar”. Parece com a mensagem mostrada acima quando se tenta uma comunicação TCP/IP. Um engano comum é esquecer de configurar o servidor para aceitar conexões TCP/IP.

Outra possibilidade é receber esta mensagem de erro ao se tentar uma comunicação através do soquete do domínio Unix com o servidor local:

```
psql: could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

-- Tradução da mensagem

```
psql: não foi possível conectar com o servidor. Arquivo ou diretório inexistente
        O servidor está executando localmente e aceitando
        conexões no soquete do domínio Unix "/tmp/.s.PGSQL.5432"?
```

A última linha é útil para verificar se o cliente está tentando se conectar ao local correto. Se existir realmente um servidor executando neste local, o núcleo da mensagem de erro será normalmente *Conexão recusada* (Connection refused) ou *Arquivo ou diretório inexistente* (No such file or directory), como mostrado (É importante perceber que neste contexto *Conexão recusada* não significa que o servidor recebeu o pedido de conexão e o recusou. Este caso produz uma mensagem diferente, conforme mostrado na Seção 19.3). Outras mensagens de erro, como *Tempo de conexão esgotado*, podem indicar problemas mais básicos, como falta de conectividade da rede.

## 16.4. Configuração em tempo de execução

Existem vários parâmetros de configuração que afetam o comportamento do sistema de banco de dados. Nesta seção é descrito como definir os parâmetros de configuração; as subseções abaixo discutem cada parâmetro em detalhe.

Não há diferença entre letras maiúsculas e minúsculas nos nomes dos parâmetros. Todo parâmetro aceita um valor de um destes quatro tipos: booleano, inteiro, ponto flutuante ou cadeia de caracteres. Os valores booleanos podem ser escritos como ON, OFF, TRUE, FALSE, YES, NO, 1 ou 0 (sem distinção entre letras maiúsculas e minúsculas), ou qualquer prefixo não ambíguo destes.

Uma forma de definir estes parâmetros é editar o arquivo `postgresql.conf`, normalmente presente no diretório de dados (O utilitário `initdb` instala uma cópia padrão neste diretório). Um exemplo de como este arquivo se parece é:

```
# Isto é um comentário
log_connections = yes
log_destination = 'syslog'
search_path = '$user, public'
```

É especificado um parâmetro por linha. O sinal de igual entre o nome e o valor é opcional. Espaços em branco não são significativos e as linhas em branco são ignoradas. O caractere jogo-da-velha (#) insere comentário em qualquer posição. Os valores dos parâmetros, que não são identificadores simples ou números, devem estar entre apóstrofos (').

O arquivo de configuração é lido novamente sempre que o processo `postmaster` recebe do sistema o sinal SIGHUP (cuja forma mais fácil de enviar é através de `pg_ctl reload`). O `postmaster` também propaga este sinal para todos os processos servidor em execução, para que as sessões em andamento também leiam os novos valores. Como alternativa, o sinal pode ser enviado diretamente para um único processo servidor. Alguns parâmetros somente podem ser definidos durante a inicialização do servidor; qualquer modificação de seus valores no arquivo de configuração será ignorada até que o servidor seja reiniciado.

A segunda forma de definir estes parâmetros de configuração é fornecê-los como opção de linha de comando para o `postmaster`, como em:

```
postmaster -c log_connections=yes -c log_destination='syslog'
```

As opções de linha de comando têm precedência sobre qualquer definição conflitante presente no arquivo `postgresql.conf`. Deve ser observado que isto significa não ser possível alterar em tempo de execução um valor fornecido como opção de linha de comando editando o arquivo `postgresql.conf`. Portanto, embora o método linha de comando possa ser conveniente, pode causar perda de flexibilidade.

Ocasionalmente, também é útil fornecer uma opção de linha de comando para apenas uma determinada sessão. Para esta finalidade pode ser utilizada a variável de ambiente `PGOPTIONS` no lado cliente:

```
env PGOPTIONS='-c geqo=off' psql
```

(Funciona para todo aplicativo cliente baseado na biblioteca libpq, e não apenas para o psql). Deve ser observado que não funciona para os parâmetros que são fixados na inicialização do servidor, ou que devem ser especificados no arquivo `postgresql.conf`.

Além disso, é possível atribuir um conjunto de definições de parâmetro a um usuário ou a um banco de dados. Sempre que uma sessão é iniciada, as definições padrão para o usuário ou para o banco de dados são carregadas. Para configurar estas definições são utilizados os comandos *ALTER USER* e *ALTER DATABASE*, respectivamente. As definições para o banco de dados têm precedência sobre qualquer definição recebida pela linha de comando do `postmaster` ou através do arquivo de configuração. Por sua vez, as definições para o usuário têm precedência sobre as definições para o banco de dados; as opções para a sessão têm precedência sobre as duas.

Alguns parâmetros podem ser alterados nas sessões SQL individuais utilizando o comando *SET* como, por exemplo:

```
SET ENABLE_SEQSCAN TO OFF;
```

Se o comando *SET* for permitido, tem precedência sobre todas as outras fontes de valor para parâmetros. Alguns parâmetros não podem ser alterados através do comando *SET*: por exemplo, parâmetros que controlam um comportamento que não pode ser alterado sem reiniciar o PostgreSQL. Além disso, alguns parâmetros podem ser alterados através dos comandos *SET* ou *ALTER* pelos superusuários, mas não pelos usuários comuns.

O comando *SHOW* permite ver o valor corrente de todos os parâmetros.

A tabela virtual `pg_settings` (descrita na Seção 41.35) também permite ver e atualizar os parâmetros em tempo de execução da sessão. Equivale ao comando *SHOW* e *SET*, mas seu uso pode ser mais conveniente, porque pode ser feita a junção com outras tabelas, ou feita a seleção utilizando a condição de seleção desejada.

### 16.4.1. Locais dos arquivos

Além do arquivo `postgresql.conf` já mencionado, o PostgreSQL utiliza outros dois arquivos de configuração, editados manualmente, para controlar a autenticação de clientes, (estes arquivos são mostrados no Capítulo 19). Por padrão todos estes três arquivos de configuração são armazenados no diretório de dados do agrupamento de bancos de dados. Os parâmetros descritos nesta subseção permitem colocar os arquivos de configuração em outro local (Fazer isto pode facilitar a administração. Em particular, geralmente é mais fácil fazer a cópia de segurança dos arquivos de configuração quando estes são mantidos separados dos arquivos do diretório de dados).

`data_directory (string)`

Especifica o diretório a ser utilizado para armazenamento dos dados. Somente pode ser definido na inicialização do servidor.

`config_file (string)`

Especifica o arquivo principal de configuração do servidor (geralmente chamado `postgresql.conf`). Somente pode ser definido na linha de comando do `postmaster`.

`hba_file (string)`

Especifica o arquivo de configuração para autenticação baseada no hospedeiro (normalmente chamado `pg_hba.conf`). Somente pode ser definido na inicialização do servidor.

`ident_file (string)`

Especifica o arquivo de configuração para a autenticação `ident` (normalmente chamado `pg_ident.conf`). Somente pode ser definido na inicialização do servidor.

`external_pid_file (string)`

Especifica o nome de um arquivo de identificação de processo adicional, que o `postmaster` deve criar para uso pelos programas de administração do servidor. Somente pode ser definido na inicialização do servidor.

Em uma instalação padrão, nenhum dos parâmetros acima é definido explicitamente. Em vez disso, o diretório de dados é especificado através da opção de linha de comando `-D`, ou através da variável de ambiente `PGDATA`, e os arquivos de configuração ficam todos no diretório de dados.

Se for desejado manter os arquivos de configuração em um local diferente do diretório de dados, então a opção de linha de comando `-D` do `postmaster`, ou a variável de ambiente `PGDATA`, deve apontar para o diretório que contém os arquivos de configuração, e o parâmetro `data_directory` deve estar definido no arquivo `postgresql.conf` (ou na linha de comando) para indicar onde o diretório de dados realmente se encontra. Deve ser observado que o parâmetro `data_directory` tem precedência sobre `-D` e sobre `PGDATA` para o local do diretório de dados, mas não para o local dos arquivos de configuração.

Se for desejado, podem ser especificados individualmente os nomes dos arquivos e seus locais utilizando os parâmetros `config_file`, `hba_file` e `ident_file`. O parâmetro `config_file` somente pode ser especificado na linha de comando do `postmaster`, mas os outros dois podem ser definidos no arquivo de configuração principal. Se, além de `data_directory`, estes outros três parâmetros forem definidos explicitamente, então não é necessário especificar `-D` ou `PGDATA`.

Quando uma destas três opções é definida, um caminho relativo é interpretado como sendo relativo ao diretório onde o `postmaster` é inicializado.

## 16.4.2. Conexões e Autenticação

### 16.4.2.1. Definições de conexão

`listen_addresses (string)`

Especifica o endereço, ou endereços, de TCP/IP onde o servidor atende as conexões dos aplicativos cliente. O valor tem a forma de uma lista de nomes de hospedeiros, ou de endereços numéricos de IP, separados por vírgula. A entrada especial `*` corresponde a todas as interfaces de IP disponíveis. Se a lista estiver vazia, o servidor não atende nenhuma interface IP e, neste caso, somente podem ser utilizados soquetes do domínio Unix para conectar ao servidor de banco de dados. O valor padrão é `localhost`, que permite serem feitas apenas conexões locais “retornantes” (`loopback`). Somente pode ser definido na inicialização do servidor.

`port (integer)`

A porta TCP onde o servidor está atendendo; 5432 por padrão. Deve ser observado que é utilizada a mesma porta em todos os endereços de IP onde o servidor atende. Somente pode ser definido na inicialização do servidor.

`max_connections (integer)`

Determina o número máximo de conexões simultâneas ao servidor de banco de dados. O valor típico é 100, mas pode ser menor se a configuração do núcleo do sistema operacional não tiver capacidade para suportar este valor (conforme determinado durante o `initdb`). Somente pode ser definido na inicialização do servidor.

O aumento deste parâmetro pode fazer com que o PostgreSQL requisiite mais memória compartilhada System V que o permitido pela configuração padrão do sistema operacional. Para obter informações sobre como ajustar estes parâmetros deve ser consultada a Seção 16.5.1, se for necessário.

`superuser_reserved_connections (integer)`

Determina o número de “encaixes de conexão” (`connection slots`), reservados para os superusuários do PostgreSQL se conectarem. Podem estar ativas até `max_connections` conexões simultâneas. Sempre que o número de conexões ativas simultâneas for igual ou maior a `max_connections` menos `superuser_reserved_connections`, somente serão aceitas novas conexões feitas por superusuários.

O valor padrão é 2. O valor deve ser menor que o valor de `max_connections`. Somente pode ser definido na inicialização do servidor.

`unix_socket_directory (string)`

Especifica o diretório do soquete do domínio Unix onde o servidor está atendendo as conexões dos aplicativos cliente. Normalmente o valor padrão é `/tmp`, mas pode ser mudado em tempo de construção. Somente pode ser definido na inicialização do servidor.

`unix_socket_group (string)`

Define o grupo dono do soquete do domínio Unix (O usuário dono do soquete é sempre o usuário que inicializa o servidor). Combinado com o parâmetro `unix_socket_permissions` pode ser utilizado como um mecanismo de controle de acesso adicional para as conexões do domínio Unix. Por padrão é uma cadeia de caracteres vazia, que utiliza o grupo padrão do usuário corrente. Somente pode ser definido na inicialização do servidor.

`unix_socket_permissions (integer)`

Define as permissões de acesso do soquete do domínio Unix. Os soquetes do domínio Unix utilizam o conjunto usual de permissões do sistema de arquivos do Unix. Para o valor deste parâmetro, é esperada uma especificação de modo numérica, na forma aceita pelas chamadas de sistema `chmod` e `umask` (Para utilizar o formato octal, como é de costume, o número deve começar por 0 (zero)).

As permissões padrão são 0777, significando que qualquer um pode se conectar. Alternativas razoáveis são 0770 (somente o usuário e o grupo, consulte também `unix_socket_group`), e 0700 (somente o usuário); deve ser observado que, na verdade, para soquetes do domínio Unix somente a permissão de escrita tem importância, não fazendo sentido conceder ou revogar permissões de leitura e de execução.

Este mecanismo de controle de acesso é independente do descrito no Capítulo 19.

Somente pode ser definido na inicialização do servidor.

`rendezvous_name (string)`

Especifica o nome de difusão (`broadcast`) Rendezvous. Por padrão é utilizado o nome do computador, especificado através de uma cadeia de caracteres vazia ("). É ignorado quando o servidor não é compilado com suporte a Rendezvous. Somente pode ser definido na inicialização do servidor.

#### 16.4.2.2. Segurança e autenticação

`authentication_timeout (integer)`

Tempo máximo, em segundos, para completar a autenticação do cliente. Se a tentativa de tornar-se cliente não completar o protocolo de autenticação nesta quantidade de tempo, o servidor derruba a conexão. Isto impede que clientes travados fiquem ocupando a conexão indefinidamente. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`. O valor padrão é 60.

`ssl (boolean)`

Habilita conexões SSL. Por favor leia a Seção 16.7 antes de utilizar este parâmetros. O valor padrão é falso. Somente pode ser definido na inicialização do servidor.

`password_encryption (boolean)`

Quando é especificada uma senha em *CREATE USER* ou *ALTER USER*, sem que seja escrito `ENCRYPTED` ou `UNENCRYPTED`, este parâmetro determina se a senha deve ser criptografada. O valor padrão é verdade (criptografar a senha).

`krb_server_keyfile (string)`

Define o local do arquivo de chave do servidor Kerberos. Para obter detalhes deve ser consultada a Seção 19.2.3 .

`db_user_namespace (boolean)`

Permite nomes de usuário por banco de dados. O valor padrão é falso.

Se o valor for verdade, os usuários devem ser criados como `nome_do_usuario@nome_bd`. Quando o `nome_do_usuario` é passado por um cliente se conectando, são anexados @ e o nome do banco de dados ao nome do usuário, então o servidor procura por este nome de usuário específico do banco de dados. Deve ser observado que, no ambiente SQL, para criar nomes de usuário contendo @ é necessário colocar o nome de usuário entre aspas.

Quando o valor deste parâmetro é verdade, ainda podem ser criados usuários globais comuns. Deve apenas ser anexado o caractere @ à especificação do nome de usuário no cliente. O caractere @ será retirado antes do nome de usuário ser procurado pelo servidor.

**Nota:** Esta funcionalidade foi criada como uma medida temporária até que seja encontrada uma solução definitiva, quando então será removida.

### 16.4.3. Consumo de recursos

#### 16.4.3.1. Memória

`shared_buffers (integer)`

Define o número de `buffers` de memória compartilhada, utilizados pelo servidor de banco de dados. O valor típico é 1000, mas pode ser menor se a configuração do núcleo do sistema operacional não tiver capacidade para suportar este valor (conforme determinado durante o `initdb`). Cada `buffer` possui 8192 bytes, a menos que seja escolhido um valor diferente para `BLCKSZ` ao construir o servidor. O valor definido deve ser pelo menos igual a 16, assim como pelo menos duas vezes o valor de `max_connections`; entretanto, normalmente é necessário definir um valor bem maior que o mínimo para obter um bom desempenho. São recomendados valores de alguns poucos milhares para instalações de produção. Somente pode ser definido na inicialização do servidor.

O aumento deste parâmetro pode fazer com que o PostgreSQL requisiite mais memória compartilhada System V que o permitido pela configuração padrão do sistema operacional. Para obter informações sobre como ajustar estes parâmetros deve ser consultada a Seção 16.5.1, se for necessário.

`work_mem (integer)`

Especifica a quantidade de memória utilizada pelas operações internas de classificação e tabelas de dispersão (`hash tables`), antes de alternar para arquivos temporários em disco. O valor é especificado em kilobytes, e o valor padrão é 1024 kilobytes (1 MB). Deve ser observado que, em uma consulta complexa, podem ser executadas várias classificações ou operações de `hash` em paralelo; cada uma podendo utilizar tanta memória quanto especificado por este parâmetro, antes de colocar os dados em arquivos temporários. Além disso, diversas sessões em execução podem estar fazendo operações de classificação simultaneamente. Portanto, a memória total utilizada pode ser várias vezes o valor de `work_mem`; é necessário ter este fato em mente ao escolher o valor. As operações de classificação são utilizadas por `ORDER BY`, `DISTINCT` e junções por mesclagem (`merge joins`). As tabelas de dispersão (`hash tables`) são utilizadas em junções de dispersão (`hash joins`), agregações baseadas em dispersão (`hash-based aggregation`), e no processamento baseado em dispersão (`hash-based processing`) de subconsultas `IN`.

`maintenance_work_mem (integer)`

Especifica a quantidade máxima de memória que pode ser utilizada pelas operações de manutenção, como `VACUUM`, `CREATE INDEX` e `ALTER TABLE ADD FOREIGN KEY`. O valor é especificado em kilobytes, e o valor padrão é 16384 kilobytes (16 MB). Uma vez que somente pode ser executada uma destas operações por vez em uma mesma sessão de banco de dados, e o servidor normalmente não possui muitas delas acontecendo ao mesmo tempo, é seguro definir este valor significativamente maior que `work_mem`. Definições maiores podem melhorar o desempenho do comando `VACUUM` e da restauração de cópias de segurança.

`max_stack_depth (integer)`

Especifica a profundidade máxima segura para a pilha de execução do servidor. A definição ideal deste parâmetro é o limite do tamanho da pilha imposto pelo núcleo do sistema operacional (conforme definido pelo comando `ulimit -s` <sup>2</sup> ou seu equivalente local, menos uma margem de segurança em torno de um megabyte. A margem de segurança é necessária porque a profundidade da pilha não é verificada em todas as rotinas do servidor, mas somente nas rotinas potencialmente recursivas chave, como avaliação de expressão. Definir o parâmetro acima do limite do núcleo significa que uma função recursiva fora de controle pode derrubar um processo servidor individual. A definição padrão é 2048 KB (dois megabytes), que é conservadoramente pequena e com pouca chance de risco de queda. Entretanto, pode ser muito pequena para permitir a execução de funções complexas.

#### 16.4.3.2. Mapa do espaço livre

`max_fsm_pages (integer)`

Define o número máximo de páginas de disco para as quais o espaço livre será acompanhado no mapa de espaço livre compartilhado. São consumidos seis bytes de memória compartilhada para cada encaixe de página. O valor definido deve ser maior que `16 * max_fsm_relations`. O valor padrão é 20000. Somente pode ser definido na inicialização do servidor.



`max_fsm_relations (integer)`

Define o número máximo de relações (tabelas e índices) para as quais o espaço livre será acompanhado no mapa de espaço livre compartilhado. São consumidos, aproximadamente, 50 bytes de memória compartilhada por cada encaixe. O valor padrão é 1000. Somente pode ser definido na inicialização do servidor.

#### 16.4.3.3. Utilização de recursos do núcleo

`max_files_per_process (integer)`

Define o número máximo permitido de arquivos abertos simultaneamente por cada subprocesso servidor. O valor padrão é 1000. Se o núcleo tiver um limite de segurança por processo, não é necessário se preocupar com esta definição. Entretanto, em algumas plataformas (notadamente a maioria dos sistemas BSD), o núcleo permite que processos individuais abram muito mais arquivos que o sistema pode suportar, quando um grande número de processos tenta abrir esta quantidade de arquivos ao mesmo tempo. Se for vista a mensagem de erro “Muitos arquivos abertos” deve-se tentar reduzir esta definição. Somente pode ser definido na inicialização do servidor.

`preload_libraries (string)`

Especifica uma ou mais bibliotecas compartilhadas a serem pré-carregadas durante a inicialização do servidor. Pode ser chamada, opcionalmente, uma função de inicialização sem parâmetros para cada biblioteca. Para especificá-la deve ser adicionado dois-pontos e o nome da função de inicialização após o nome da biblioteca. Por exemplo '`$libdir/minha_bib:minha_bib_inic`' faz com que `minha_bib` seja pré-carregada e `minha_bib_inic` seja executada. Para carregar mais de uma biblioteca, os nomes devem ser separados por vírgula.

Se `minha_bib` ou `minha_bib_inic` não for encontrada, a inicialização do servidor não será bem-sucedida.

As bibliotecas de linguagem procedural do PostgreSQL são pré-carregadas desta maneira, usualmente utilizando a sintaxe '`$libdir/plXXX:plXXX_init`', onde `XXX` é `pgsql`, `perl`, `tcl` ou `python`.

Fazendo a pré-carga da biblioteca compartilhada (e a inicializando, se for aplicável), ganha-se o tempo de inicialização da biblioteca quando a biblioteca é utilizada pela primeira vez. Entretanto, pode aumentar ligeiramente o tempo para inicializar cada novo processo servidor, mesmo que o processo nunca utilize a biblioteca. Portanto, este parâmetro só é recomendado para bibliotecas utilizadas pela maioria das sessões.

#### 16.4.3.4. Retardo do VACUUM baseado no custo

Durante a execução dos comandos *VACUUM* e *ANALYZE*, o sistema mantém um contador interno que acompanha o custo estimado das várias operações de E/S que são realizadas. Quando o custo acumulado atinge um limite (especificado por `vacuum_cost_limit`), o processo que está realizando a operação adormece por um tempo (especificado por `vacuum_cost_delay`). Depois o contador é reiniciado e a execução continua.

O objetivo desta funcionalidade é permitir que os administradores reduzam o impacto na E/S gerado por estes comandos sobre as atividades de banco de dados simultâneas. Existem diversas situações onde não é muito importante que comandos como *VACUUM* e *ANALYZE* terminem rapidamente; entretanto, geralmente é muito importante que estes comandos não interfiram significativamente com a capacidade do sistema de realizar outras operações de banco de dados. O retardo do *VACUUM* baseado no custo fornece uma maneira dos administradores atingirem este objetivo.

Esta funcionalidade está desabilitada por padrão. Para ser habilitada, o parâmetro `vacuum_cost_delay` deve ser definido com um valor diferente de zero.

`vacuum_cost_delay (integer)`

A quantidade de tempo, em milissegundos, que o processo adormece quando o custo limite é excedido. O valor padrão é 0, que desabilita a funcionalidade de retardo do *VACUUM* baseado no custo. Valores positivos habilitam o retardo do *VACUUM* baseado no custo. Deve ser observado que em muitos sistemas a resolução efetiva de adormecimento é de 10 milissegundos; definir `vacuum_cost_delay` com um valor que não é múltiplo de 10, pode ter o mesmo resultado que definir com o próximo múltiplo de 10 maior que o valor especificado.

`vacuum_cost_page_hit (integer)`

O custo estimado para executar o *VACUUM* em um `buffer` encontrado no `buffer cache` compartilhado. Representa o custo para bloquear o `buffer pool`, examinar a tabela `hash` compartilhada, e varrer o conteúdo da página. O valor padrão é 1.

`vacuum_cost_page_miss (integer)`

O custo estimado para executar o VACUUM em um buffer que precisa ser lido no disco. Representa o esforço para bloquear o buffer pool, examinar a tabela hash compartilhada, ler o bloco desejado no disco e varrer o seu conteúdo. O valor padrão é 10.

`vacuum_cost_page_dirty (integer)`

O custo estimado cobrado quando o VACUUM modifica um bloco que foi limpo anteriormente. Representa a E/S adicional necessária para descarregar no disco novamente um bloco que foi limpo anteriormente. O valor padrão é 20.

`vacuum_cost_limit (integer)`

O custo acumulado que faz com que o processo executando o VACUUM adormeça. O valor padrão é 200.

**Nota:** Existem determinadas operações que prendem bloqueios críticos e que devem, portanto, terminar o mais rápido possível. O retardo do VACUUM baseado no custo não ocorre durante este tipo de operação. Portanto, é possível que o custo acumule bem acima do limite especificado. Nestes casos, para evitar retardos longos sem utilidade, o retardo real é calculado como  $\text{vacuum\_cost\_delay} * \text{accumulated\_balance} / \text{vacuum\_cost\_limit}$  sendo no máximo igual a  $\text{vacuum\_cost\_delay} * 4$ .

#### 16.4.3.5. Escrita em segundo plano

A partir do PostgreSQL 8.0 passa a existir um servidor em separado, chamado de *escritor de segundo plano* (background writer), cuja única função é escrever buffers de memória compartilhada “sujos” (dirty) no disco. O objetivo é fazer com que raramente, ou nunca, os processos servidor que tratam os comandos dos usuários tenham que aguardar uma escrita em disco, porque o escritor de segundo plano já terá feito esta escrita. Esta funcionalidade também reduz a degradação de desempenho associada aos pontos de controle. O escritor de segundo plano escreve continuamente as páginas no disco, de modo que no momento do ponto de controle somente é necessário escrever algumas poucas páginas, em vez de ser necessário escrever uma grande quantidade de buffers sujos como acontecia anteriormente. Entretanto, existe um aumento global na carga de E/S, porque anteriormente uma página suja várias vezes era escrita apenas uma vez no disco no período de um ponto de controle, enquanto agora o escritor de segundo plano pode escrever esta página no disco várias vezes no mesmo período. Na maioria das situações se prefere uma baixa carga contínua do que um pico periódico. Os parâmetros vistos nesta seção podem ser utilizados para ajustar o comportamento conforme as necessidades locais.<sup>3</sup>

`bgwriter_delay (integer)`

Especifica o retardo entre rodadas de atividade para escritor de segundo plano. A cada rodada o escritor escreve uma quantidade de buffers sujos (controlado pelos parâmetros mostrados a seguir). Os buffers selecionados são sempre os usados menos recentemente entre os buffers sujos atuais. Em seguida adormece por `bgwriter_delay` milissegundos e repete a operação. O valor padrão é 200. Deve ser observado que em muitos sistemas a resolução efetiva do adormecimento é de 10 milissegundos; definir `bgwriter_delay` com um valor que não é múltiplo de 10, pode ter o mesmo resultado que definir com o próximo múltiplo de 10 maior que o valor especificado. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

`bgwriter_percent (integer)`

A cada rodada não será escrito mais que esta percentagem de buffers sujos no momento (as frações são arredondadas para o próximo número inteiro de buffers acima). O valor padrão é 1. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

`bgwriter_maxpages (integer)`

A cada rodada não será escrito mais que esta quantidade de buffers sujos. O valor padrão é 100. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

A utilização de valores menores para `bgwriter_percent` e `bgwriter_maxpages` reduz a carga extra de E/S causada pelo escritor de segundo plano, mas deixa mais trabalho a ser feito no ponto de controle. Para reduzir o pico de carga nos pontos de controle estes valores devem ser aumentados. Para desabilitar inteiramente o escritor de segundo plano os parâmetros `bgwriter_percent` e/ou `bgwriter_maxpages` devem ser definidos iguais a zero.

### 16.4.4. Log de escrita prévia (WAL)

Consulte também a Seção 25.2 para obter detalhes sobre o ajuste do WAL.

#### 16.4.4.1. Definições

`fsync (boolean)`

Se o valor deste parâmetro for `true`, o servidor PostgreSQL utilizará a chamada de sistema `fsync()` em vários lugares para ter certeza que as atualizações estão fisicamente escritas no disco. Isto garante que o agrupamento de bancos de dados vai ser recuperado em um estado consistente após um problema de máquina ou do sistema operacional.

Entretanto, a utilização de `fsync()` produz uma degradação de desempenho: quando a transação é efetivada, o PostgreSQL tem de aguardar o sistema operacional descarregar o log de escrita prévia no disco. Quando `fsync` está desabilitado, o sistema operacional pode desempenhar da sua melhor maneira a “bufeização”, ordenação e retardo na escrita. Isto pode produzir uma melhora significativa no desempenho. Porém, no caso de uma queda do sistema, podem ser perdidos, em parte ou por inteiro, os resultados das poucas últimas transações efetivadas. No pior caso, os dados podem ficar corrompidos de uma forma irrecuperável (Para esta situação a queda do servidor de banco de dados *não* é um fator de risco, somente há risco dos dados ficarem corrompidos no caso de queda do sistema operacional).

Devido aos riscos envolvidos não existe uma definição universalmente aceita para `fsync`. Alguns administradores sempre desabilitam `fsync`, enquanto outros só desabilitam para cargas de dado pesadas, onde claramente existe um ponto de recomeço se algo de errado acontecer, enquanto outros administradores sempre deixam `fsync` habilitado. O valor padrão para `fsync` é habilitado, para obter o máximo de confiabilidade. Havendo confiança no sistema operacional, na máquina, e nos utilitários que acompanham (ou na bateria de reserva), pode-se levar em consideração desabilitar `fsync`.

Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

`wal_sync_method (string)`

Método utilizado para obrigar a colocar as atualizações do WAL no disco. Os valores possíveis são `fsync` (chamada à função `fsync()` a cada efetivação), `fdatasync` (chamada à função `fdatasync()` a cada efetivação), `open_sync` (escreve os arquivos WAL com a opção `O_SYNC` da função `open()`), e `open_datasync` (escreve arquivos WAL com a opção `O_DSYNC` do `open()`). Nem todas estas opções estão disponíveis em todas as plataformas. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

`wal_buffers (integer)`

Número de `buffers` de página de disco alocados na memória compartilhada para os dados do WAL. O valor padrão é 8. Esta definição somente precisa ser grande o suficiente para conter a quantidade de dados do WAL gerados por uma transação típica. Somente pode ser definido na inicialização do servidor.

`commit_delay (integer)`

Retardo entre a escrita do registro de efetivação no `buffer` do WAL e a descarga do `buffer` no disco, em microssegundos. Um retardo maior que zero permite que seja feita apenas uma chamada de sistema `fsync()` para várias transações efetivadas, se a carga do sistema for alta o suficiente para que outras transações fiquem prontas para efetivar dentro do intervalo especificado. Entretanto, este retardo é simplesmente desperdício se nenhuma outra transação ficar pronta para efetivar. Portanto, o retardo somente é realizado se pelo menos outras `commit_siblings` transações estiverem ativas no instante que o processo servidor escrever seu registro de efetivação. O valor padrão é zero (nenhum retardo).

`commit_siblings (integer)`

Número mínimo de transações simultâneas abertas requerido para realizar o retardo `commit_delay`. Um valor maior torna mais provável que pelo menos uma outra transação fique pronta para efetivar durante o período do retardo. O valor padrão é 5.

#### 16.4.4.2. Pontos de controle

`checkpoint_segments (integer)`

Distância máxima entre pontos de controle automático do WAL, em segmentos de arquivo de `log` (cada segmento possui normalmente 16 megabytes). O valor padrão é 3. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

`checkpoint_timeout (integer)`

Tempo máximo, em segundos, entre pontos de controle automáticos do WAL. O valor padrão é 300 segundos. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

`checkpoint_warning (integer)`

Escreve uma mensagem no `log` do servidor caso ocorra um ponto de controle, causado pelo enchimento dos arquivos de segmento de ponto de controle, em um tempo menor que este número de segundos. O valor padrão é 30 segundos. Zero desabilita a advertência.

#### 16.4.4.3. Cópia de segurança

`archive_command (string)`

O comando a ser passado para o interpretador de comandos para executar a cópia de segurança de um segmento da série de arquivos do WAL quando este é completado. Se for uma cadeia de caracteres vazia (que é o valor padrão), a cópia de segurança do WAL é desabilitada. Todo `%p` presente na cadeia de caracteres é substituído pelo caminho absoluto do arquivo cuja cópia de segurança será feita, e todo `%f` é substituído apenas pelo nome do arquivo. Deve ser utilizado `%%` para incluir o caractere `%` na linha de comando. Para obter mais informações deve ser consultada a Seção 22.3.1. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`. file.

É importante que o comando retorne um status de saída igual a zero se, e somente se, for bem-sucedido. Exemplos:

```
archive_command = 'cp "%p" /mnt/server/archivedir/"%f"'
archive_command = 'copy "%p" /mnt/server/archivedir/"%f"' # Windows
```

### 16.4.5. Planejamento de comando

#### 16.4.5.1. Configuração do método de planejamento

Estes parâmetros de configuração fornecem um método rudimentar para influenciar os planos de comando escolhidos pelo otimizador de comandos. Se o plano padrão escolhido pelo otimizador para um determinado comando não for o ótimo, uma solução temporária pode ser obtida utilizando um destes parâmetros de configuração para forçar o otimizador a escolher um plano diferente. Entretanto, desabilitar uma destas definições permanentemente dificilmente é uma boa idéia. Outras formas de melhorar a qualidade dos planos escolhidos pelo otimizador são o ajuste das *Constantes de custo do planejador*, a execução do comando *ANALYZE* com mais frequência, o aumento do valor do parâmetro de configuração `default_statistics_target`, e o aumento da quantidade de estatísticas coletadas para colunas específicas utilizando o comando `ALTER TABLE SET STATISTICS`.

`enable_hashagg (boolean)`

Habilita ou desabilita o uso pelo planejador de comandos dos planos do tipo agregação por hash. O valor padrão é habilitado.

`enable_hashjoin (boolean)`

Habilita ou desabilita o uso pelo planejador de comandos dos planos do tipo junção por hash. O valor padrão é habilitado.

`enable_indexscan (boolean)`

Habilita ou desabilita o uso pelo planejador de comandos dos planos do tipo varredura de índice. O valor padrão é habilitado.

`enable_mergejoin (boolean)`

Habilita ou desabilita o uso pelo planejador de comandos dos planos do tipo junção por mesclagem. O valor padrão é habilitado.

`enable_nestloop` (boolean)

Habilita ou desabilita o uso pelo planejador de comandos dos planos do tipo junção por laço aninhado. Não é possível suprimir junções por laço aninhado inteiramente, mas tornar o valor deste parâmetro igual a `false` desestimula a utilização deste método pelo planejador, quando há outro método disponível. O valor padrão é habilitado.

`enable_seqscan` (boolean)

Habilita ou desabilita o uso pelo planejador de comandos dos planos do tipo varredura sequencial. Não é possível suprimir as varreduras sequenciais inteiramente, mas tornar o valor deste parâmetro igual a `false` desestimula a utilização deste método pelo planejador, quando há outro método disponível. O valor padrão é habilitado.

`enable_sort` (boolean)

Habilita ou desabilita o uso pelo planejador de comandos dos passos de classificação explícita pelo planejador de comandos. Não é possível suprimir as classificações explícitas inteiramente, mas tornar o valor deste parâmetro igual a `false` desestimula a utilização deste método pelo planejador, quando há outro método disponível. O valor padrão é habilitado.

`enable_tidscan` (boolean)

Habilita ou desabilita o uso pelo planejador de comandos dos planos do tipo varredura TID. O valor padrão é habilitado.

#### 16.4.5.2. Constantes de custo do planejador

**Nota:** Infelizmente, não existe nenhum método bem definido para determinar os valores ideais para a família de variáveis de “custo” mostradas abaixo. Incentivamos que sejam feitas experiências e compartilhadas as descobertas.

`effective_cache_size` (floating point)

Define o tamanho efetivo presumido pelo planejador acerca do `cache` de disco disponível para uma única varredura de índice. É um dos elementos usados na estimativa do custo de utilização de um índice; um valor maior torna mais provável a utilização de uma varredura de índice, enquanto um valor menor torna mais provável a utilização de uma varredura sequencial. Ao definir este valor devem ser considerados os `buffers` de memória compartilhada do PostgreSQL, e a parcela do `cache` de disco do núcleo que será utilizada pelos arquivos de dado do PostgreSQL. Também deve ser levado em consideração o número esperado de comandos simultâneos que utilizam índices diferentes, uma vez que estes têm de compartilhar o espaço disponível. Este parâmetro não tem efeito sobre o tamanho da memória compartilhada alocada pelo PostgreSQL, nem reserva `cache` de disco do núcleo; é usado apenas para finalidades de estimativa. O valor é medido em páginas de disco, normalmente com 8192 bytes cada. O valor padrão é 1000.

`random_page_cost` (floating point)

Define a estimativa do planejador de comandos do custo da busca não sequencial de uma página de disco. É medido como um múltiplo do custo da busca sequencial de uma página. Um valor maior torna mais provável a utilização de uma varredura sequencial, enquanto um valor menor torna mais provável a utilização de uma uma varredura de índice. O valor padrão é 4.

`cpu_tuple_cost` (floating point)

Define a estimativa do planejador de comandos do custo de processamento de cada linha durante o comando. É medido como uma fração do custo da busca de uma página sequencial. O valor padrão é 0.01.

`cpu_index_tuple_cost` (floating point)

Define a estimativa do planejador de comandos do custo de processamento de cada linha de índice durante a varredura do índice. Medido como uma fração do custo da busca de uma página sequencial. O valor padrão é 0.001.

`cpu_operator_cost` (floating point)

Define a estimativa do planejador de comandos do custo de processamento de cada operador na cláusula `WHERE`. É medido como uma fração do custo da busca de uma página sequencial. O valor padrão é 0.0025.

### 16.4.5.3. Otimização genética de comandos

`gego` (boolean)

Habilita ou desabilita a otimização genética de comandos, que é um algoritmo que tenta realizar um planejamento de comandos sem uma busca exaustiva. O valor padrão é habilitado. A variável `gego_threshold` permite uma forma mais granular para desabilitar GEQO para certas classes de comandos.

`gego_threshold` (integer)

A otimização genética de comandos é utilizada para planejar comandos com pelo menos esta quantidade de itens envolvidos na cláusula `FROM` (Deve ser observado que uma construção de `JOIN` externa conta como apenas um item da cláusula `FROM`). O valor padrão é 12. Para comandos simples geralmente é melhor utilizar o planejamento determinístico exaustivo, mas para comandos com muitas tabelas o planejamento determinístico leva muito tempo.

`gego_effort` (integer)

Controla o equilíbrio entre o tempo de planejamento e a eficiência do planejamento do comando no GEQO. O valor desta variável deve ser um número inteiro no intervalo de 1 a 10. O valor padrão é 5. Valores maiores aumentam o tempo gasto para fazer o planejamento do comando, mas também aumentam a chance de ser escolhido um plano de comando eficiente.

Na verdade, a variável `gego_effort` não faz nada diretamente; é utilizada apenas para calcular os valores padrão para as outras variáveis que influenciam o comportamento do GEQO (descritas abaixo). Caso se prefira, os demais parâmetros podem ser definidos manualmente.

`gego_pool_size` (integer)

Controla o tamanho da amostra (`pool size`) utilizado pelo GEQO. O tamanho da amostra é o número de indivíduos na população genética. Deve ser maior ou igual 2, e os valores úteis estão tipicamente no intervalo de 100 a 1000. Se for igual a zero (a definição padrão), então um valor padrão adequado é escolhido tomando por base `gego_effort` e o número de tabelas no comando.

`gego_generations` (integer)

Controla o número de gerações utilizado pelo GEQO. As gerações especificam o número de interações do algoritmo. Deve ser maior ou igual a 1, e os valores úteis estão no mesmo intervalo do tamanho da amostra. Se for definido igual a zero (a definição padrão), então um valor padrão adequado é escolhido tomando por base `gego_pool_size`.

`gego_selection_bias` (floating point)

Controla a tendência da seleção (`selection bias`) utilizada pelo GEQO. A tendência da seleção é a pressão seletiva dentro da população. Os valores podem estar entre 1.50 e 2.00; este último é o valor padrão.

### 16.4.5.4. Outras opções do planejador

`default_statistics_target` (integer)

Define a quantidade padrão de estatísticas para as colunas das tabelas que não possuem uma quantidade específica definida através do comando `ALTER TABLE SET STATISTICS`. Valores maiores aumentam o tempo necessário para executar o comando `ANALYZE`, mas podem melhorar a qualidade das estimativas do planejador. O valor padrão é 10. Para obter informações adicionais sobre a utilização das estatísticas pelo planejador de comandos do PostgreSQL deve ser consultada a Seção 13.2.

`from_collapse_limit` (integer)

O planejador incorpora as subconsultas nas consultas superiores se a lista `FROM` resultante não tiver mais do que esta quantidade de itens. Valores menores reduzem o tempo de planejamento, mas podem levar a planos de consulta inferiores. O valor padrão é 8. Geralmente é razoável manter este valor abaixo de `gego_threshold`.

`join_collapse_limit` (integer)

O planejador reescreve construções `JOIN` internas explícitas em lista de itens da cláusula `FROM`, sempre que resultar em uma lista com não mais do que esta quantidade de itens. Antes do PostgreSQL 7.4 as junções especificadas através da construção `JOIN` nunca eram reorganizadas pelo planejador. Depois o planejador foi melhorado para que as junções internas escritas desta forma pudessem ser reordenadas; este parâmetro de configuração controla até que ponto esta reordenação é realizada.

**Nota:** Atualmente a ordem das junções externas, especificadas através das construções `JOIN`, nunca é ajustada pelo planejador; portanto `join_collapse_limit` não tem efeito sobre este comportamento. O planejador poderá ser melhorado para reordenar algumas classes de junção externa em uma versão futura do PostgreSQL.

O padrão é definir esta variável com o mesmo valor de `from_collapse_limit`, o que é apropriado para a maioria dos usos. Definir o valor igual a 1 impede a reordenação dos `JOINS` internos. Portanto, a ordem de junção especificada no comando será a ordem real pela qual as relações serão juntadas. O otimizador de comandos nem sempre escolhe a ordem de junção ótima; os usuários avançados podem decidir definir o valor desta variável igual a 1 temporariamente e, então, especificar a ordem de junção desejada explicitamente. Uma outra consequência de definir o valor desta variável igual a 1 é fazer com que o planejador de comandos se comporte de forma mais parecida com o planejador de comandos do PostgreSQL 7.3, o que alguns usuários podem achar útil por motivo de compatibilidade com versões anteriores.

Definir esta variável com um valor entre 1 e `from_collapse_limit` pode ser útil para equilibrar o tempo de planejamento versus a qualidade de plano escolhido. (os valores maiores produzem planos melhores).

## 16.4.6. Relato e registro de erros

### 16.4.6.1. Onde registrar

`log_destination(string)`

O PostgreSQL dispõe de vários métodos para registrar as mensagens do servidor, incluindo `stderr` e `syslog`. No Windows também há suporte para `eventlog`. Este parâmetro é definido através de uma lista de destinos desejados para registrar as mensagens, separados por vírgula. O padrão é registrar apenas em `stderr`. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

`redirect_stderr(boolean)`

Permite que as mensagens enviadas para `stderr` sejam capturadas e redirecionadas para os arquivos de registro (`log`). Combinada com `stderr` esta opção geralmente é mais útil que registrar em `syslog`, uma vez que alguns tipos de mensagem podem não aparecer na saída para `syslog`; um exemplo comum é uma mensagem de falha do ligador dinâmico (`dynamic-linker`). Somente pode ser definido na inicialização do servidor.

`log_directory(string)`

Quando `redirect_stderr` está habilitado, este parâmetro determina o diretório onde os arquivos de registro serão criados. Pode ser especificado como um caminho absoluto, ou como um caminho relativo ao diretório de dados do agrupamento. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

`log_filename(string)`

Quando `redirect_stderr` está habilitado, este parâmetro define os nomes dos arquivos de registro criados. O valor é tratado como um padrão para a função `strftime`,<sup>4</sup> portanto podem ser utilizados os escapes de `%` para especificar nomes de arquivos que variam com o tempo. Se não houver nenhum escape de `%` presente, o PostgreSQL anexa a época do momento de abertura do arquivo de registro ao nome do arquivo. Por exemplo, se `log_filename` for igual a `server_log`, então o nome escolhido para o arquivo será `server_log.1093827753` para um registro começando em “Dom Ago 29 19:02:33 2004 MST”. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

`log_rotation_age(integer)`

Quando `redirect_stderr` está habilitado, este parâmetro determina o tempo de vida máximo de um determinado arquivo de registro. Após ter decorrido esta quantidade de minutos, é criado um novo arquivo de registro. Definir o valor como zero desabilita a criação de novos arquivos de registro baseado no tempo. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

`log_rotation_size(integer)`

Quando `redirect_stderr` está habilitado, este parâmetro determina o tamanho máximo de um arquivo de registro individual. Após esta quantidade de kilobytes ter sido lançada no arquivo de registro, é criado um novo arquivo de registro. Definir o valor como zero desabilita a criação de novos arquivos de registro baseado no tamanho. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

`log_truncate_on_rotation` (boolean)

Quando `redirect_stderr` está habilitado, este parâmetro faz com que o PostgreSQL trunque (sobrescreva), em vez de anexar, um arquivo de registro com o mesmo nome. Entretanto, o truncamento somente ocorre quando está sendo aberto um novo arquivo devido a rotação baseada no tempo, e não durante a inicialização ou rotação baseada no tamanho. Quando está desabilitado, os arquivos pré-existentes são anexados em todos os casos. Por exemplo, utilizar este parâmetro em combinação com um valor para `log_filename`, como `postgresql-%H.log`, resulta na geração de arquivos de registro de vinte e quatro horas, reescritos ciclicamente. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

Exemplo: Para manter registro dos últimos 7 dias, com um arquivo de registro por dia, chamados `server_log.Mon`, `server_log.Tue`, etc., e reescrever automaticamente o registro da semana anterior com o registro da semana corrente, `log_filename` deve ser definido como `server_log.%a`,<sup>5</sup> `log_truncate_on_rotation` como `true`, e `log_rotation_age` como 1440.

Exemplo: Para manter registros de 24 horas, um arquivo de registro por hora, mas também rotacionando mais cedo se o tamanho do arquivo de registro exceder 1GB, `log_filename` deve ser definido como `server_log.%H%M`,<sup>6 7</sup> `log_truncate_on_rotation` como `true`, `log_rotation_age` como 60, e `log_rotation_size` como 1000000. A inclusão de `%M` em `log_filename` faz com que toda rotação causada pelo tamanho do arquivo use um nome de arquivo diferente do nome de arquivo inicial da hora.

`syslog_facility` (string)

Quando o registro através de `syslog`<sup>8</sup> está habilitado, este parâmetro determina a “facilidade” do `syslog` a ser utilizada. Pode ser escolhido `LOCAL0`, `LOCAL1`, `LOCAL2`, `LOCAL3`, `LOCAL4`, `LOCAL5`, `LOCAL6`, `LOCAL7`; o padrão é `LOCAL0`. Também deve ser consultada a documentação do processo `syslog` do sistema. Somente pode ser definido na inicialização do servidor.

`syslog_ident` (string)

Quando o registro via `syslog` está habilitado, este parâmetro determina o nome de programa utilizado para identificar as mensagens do PostgreSQL nos registros do `syslog`. O valor padrão é `postgres`. Somente pode ser definido na inicialização do servidor.

#### 16.4.6.2. Quando registrar

`client_min_messages` (string)

Controla que níveis de mensagem são enviadas para o cliente. Os valores válidos são `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `LOG`, `NOTICE`, `WARNING` e `ERROR`. Cada nível inclui todos os níveis que o seguem. Quanto mais para o final estiver o nível, menos mensagens são enviadas. O valor padrão é `NOTICE`. Deve ser observado que aqui `LOG` possui uma situação relativa diferente da que tem em `log_min_messages`.

`log_min_messages` (string)

Controla que níveis de mensagem são escritas no `log` do servidor. Os valores válidos são `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL` e `PANIC`. Cada nível inclui todos os níveis que o seguem. Quanto mais para o final estiver o nível, menos mensagens são enviadas. O valor padrão é `NOTICE`. Deve ser observado que aqui `LOG` possui uma situação relativa diferente da que tem em `client_min_messages`. Somente os superusuários podem aumentar esta opção.

`log_error_verbosity` (string)

Controla a quantidade de detalhes escritos no `log` do servidor para cada mensagem que é registrada. Os valores válidos são `TERSE` (sucinto), `DEFAULT` e `VERBOSE`, cada um adicionando mais campos às mensagens escritas. Somente os superusuários podem alterar esta definição.

`log_min_error_statement` (string)

Controla se o comando SQL causador da condição de erro também será registrado no `log` do servidor. Todas as declarações SQL causadoras de um erro do nível especificado, ou de nível mais alto, são registradas. O valor padrão é `PANIC` (tornando de fato esta funcionalidade desabilitada para o uso normal). Os valores válidos são `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `FATAL` e `PANIC`. Por exemplo, se for definido como `ERROR` então todas as declarações SQL causadoras de erro, erros fatais ou pânico serão registradas. Habilitar esta



opção pode ser útil para encontrar a origem de qualquer erro que apareça no log do servidor. Somente os superusuários podem alterar esta definição.

`log_min_duration_statement (integer)`

Define o tempo de execução mínimo da declaração (em milissegundos) para que a declaração seja registrada. Todas as declarações SQL cujo tempo de execução for igual ou superior ao tempo especificado serão registradas juntamente com sua duração. Definir como zero registra todos os comandos e sua duração. O valor -1 (o padrão) desabilita esta funcionalidade. Por exemplo, se for definido como 250 então todas as declarações SQL com tempo de execução de 250ms ou superior serão registradas. Habilitar esta opção pode ser útil para encontrar comandos não otimizados nos aplicativos. Somente os superusuários podem alterar esta definição.

`silent_mode (boolean)`

Executa o servidor em silêncio. Quando o valor deste parâmetro é definido como `true`, o servidor executa automaticamente em segundo plano, e se desvincula do terminal controlador (o mesmo efeito da opção `-s` do `postmaster`). A saída padrão e o erro padrão do servidor são redirecionadas para `/dev/null` e, portanto, todas as mensagens enviadas para as mesmas são perdidas. A menos que seja habilitado registrar em `syslog`, ou que `redirect_stderr` esteja habilitado, a utilização desta opção é desencorajada, porque torna impossível ver as mensagens de erro.

Abaixo segue a relação dos vários níveis de severidade de mensagem utilizados nestas definições:

`DEBUG[1-5]`

Fornece informações para uso pelos desenvolvedores.

`INFO`

Fornece informações requisitadas implicitamente pelo usuário como, por exemplo, durante `VACUUM VERBOSE`.

`NOTICE`

Fornece informações que podem ser úteis para os usuários como, por exemplo, o truncamento de identificadores longos e a criação de índices como parte das chaves primárias.

`WARNING`

Fornece advertências para o usuário como, por exemplo, `COMMIT` fora do bloco de transação.

`ERROR`

Relata o erro que fez com que o comando corrente fosse interrompido.

`LOG`

Relata informações de interesse dos administradores como, por exemplo, atividade de ponto de controle.

`FATAL`

Relata o erro que fez com que a sessão corrente fosse interrompida.

`PANIC`

Relata o erro que fez com que todas as sessões fossem interrompidas.

### 16.4.6.3. O que registrar

`debug_print_parse (boolean)`

`debug_print_rewritten (boolean)`

`debug_print_plan (boolean)`

`debug_pretty_print (boolean)`

Estes parâmetros habilitam a emissão de várias saídas de depuração. Para cada comando executado, imprimem a árvore de análise resultante, a saída do reescritor de comando, ou o plano de execução. `debug_pretty_print` introduz recuos na mensagem mostrada, produzindo um formato de saída mais legível, mas muito mais longo. `client_min_messages` e `log_min_messages` devem estar em `DEBUG1` ou abaixo, para ser enviada a saída para o log do cliente ou do servidor, respectivamente. Estes parâmetros estão desabilitados por padrão.

`log_connections` (boolean)

Gera uma linha para o log do servidor detalhando cada conexão bem-sucedida. O valor padrão é desabilitado embora, provavelmente, seja muito útil. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

`log_disconnections` (boolean)

Gera uma linha para o log do servidor semelhante à gerada por `log_connections`, mas no término da sessão, incluindo a duração da sessão. Está desabilitado por padrão. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

`log_duration` (boolean)

Faz com que seja registrada a duração de toda declaração completada que satisfaz `log_statement`. Quando esta opção é utilizada, se syslog não estiver sendo utilizado, recomenda-se que o PID ou o ID da sessão seja registrado utilizando `log_line_prefix`, para que seja vinculada a declaração à duração utilizando o ID do processo ou o ID da sessão. O valor padrão é desabilitado. Somente os superusuários podem alterar esta definição.

`log_line_prefix` (string)

O valor deste parâmetro é uma cadeia de caracteres no estilo da função `printf`, mostrada no começo de cada linha de registro. O valor padrão é uma cadeia de caracteres vazia. Cada escape reconhecido é substituído conforme mostrado abaixo — qualquer outra coisa que se pareça com um escape é ignorada. Os demais caracteres são copiados literalmente para a linha de registro. Alguns escapes são reconhecidos apenas pelos processos de sessão, não se aplicando a processos em segundo plano como o postmaster. Syslog produz seu próprio carimbo do tempo e informação do ID do processo e, portanto, provavelmente não será desejado utilizar estes escapes quando syslog é utilizado. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

Escape	Efeito	Apenas de sessão
<code>%u</code>	Nome do usuário.	sim
<code>%d</code>	Nome do banco de dados.	sim
<code>%r</code>	Nome do hospedeiro remoto ou endereço de IP, e porta remota.	sim
<code>%p</code>	ID do processo.	não
<code>%t</code>	Carimbo do tempo.	não
<code>%i</code>	Marca do comando: O comando que gerou a linha de registro.	sim
<code>%c</code>	ID da sessão: O identificador único de cada sessão. Números hexadecimais de 2 a 4 bytes (sem zeros à frente), separados por ponto. Os números são o momento de início da sessão e o ID do processo, portanto também pode ser utilizado como uma maneira de registrar estes itens com economia de espaço.	sim
<code>%l</code>	O número da linha de registro para cada processo, começando por 1.	não
<code>%s</code>	Carimbo do tempo do início da sessão.	sim
<code>%x</code>	ID da transação.	sim
<code>%q</code>	Não produz nenhuma saída, mas informa aos processos que não são de sessão para pararem neste ponto da cadeia de caracteres. Ignorado pelos processos de sessão.	não
<code>%%</code>	O literal <code>%</code>	não

`log_statement (string)`

Controla quais declarações SQL são registradas. Os valores válidos são `none`, `ddl`, `mod` e `all`. `ddl` registra todos os comandos de definição de dados, como `CREATE`, `ALTER` e `DROP`. `mod` registra todos as instruções de `ddl`, mais `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` e `COPY FROM`. Também são registradas as instruções `PREPARE` e `EXPLAIN ANALYZE`, se os comandos contidos nestas instruções forem do tipo apropriado.

O valor padrão é `none`. Somente os superusuários podem alterar esta definição.

**Nota:** A declaração `EXECUTE` não é considerada como sendo uma declaração de `ddl` ou `mod`. Quando é registrada somente é relatado o nome da declaração preparada, e não a declaração preparada inteira.

Quando uma função é definida utilizando a linguagem do lado servidor PL/pgSQL, todos os comandos executados pela função são registrados somente na primeira vez que a função é chamada por uma determinada sessão. Isto se deve ao fato da linguagem PL/pgSQL manter um `cache` dos planos de comando produzidos pelas declarações SQL na função.

`log_hostname (boolean)`

Por padrão as mensagens de registro de conexão mostram somente o endereço de IP do hospedeiro se conectando. Habilitar esta opção faz com que seja registrado o nome do hospedeiro também. Deve ser observado que, dependendo da configuração da resolução de nome de hospedeiro, pode ser imposta uma penalidade de desempenho não desprezível. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.

## 16.4.7. Estatísticas de tempo de execução

### 16.4.7.1. Monitoramento das estatísticas

`log_statement_stats (boolean)`

`log_parser_stats (boolean)`

`log_planner_stats (boolean)`

`log_executor_stats (boolean)`

Para cada comando, são registradas estatísticas de desempenho do respectivo módulo no `log` do servidor. É um instrumento rudimentar para traçar um perfil. `log_statement_stats` relata estatísticas totais da declaração, enquanto os demais relatam estatísticas por módulo. `log_statement_stats` não pode ser habilitado junto com qualquer outro parâmetro por-módulo. Todos estes parâmetros estão desabilitados por padrão. Somente os superusuários podem alterar estas definições.

### 16.4.7.2. Coletor de estatísticas de comando e índice

`stats_start_collector (boolean)`

Controla se o servidor deve inicializar o subprocesso coletor de estatísticas. Está habilitado por padrão, mas pode ser desabilitado quando se sabe que não há interesse em coletar estatísticas. Somente pode ser definido na inicialização do servidor.

`stats_command_string (boolean)`

Habilita a coleta de estatísticas para o comando executando no momento em cada sessão, junto com o momento em que o comando começou a executar. O valor padrão é desabilitado. Deve ser observado que, mesmo quando está habilitado, esta informação não é visível por todos os usuários, mas somente pelos superusuários e o usuário dono da sessão sendo relatada; portanto, não deve representar um risco à segurança. Este dado pode ser acessado através da visão do sistema `pg_stat_activity`; para obter mais informações deve ser consultado o Capítulo 23.

`stats_block_level (boolean)`

Habilita a coleta de estatísticas no nível de bloco da atividade do banco de dados. O valor padrão é desabilitado. Se estiver habilitado, os dados produzidos podem ser acessados através da família de visões do sistema `pg_stat` e `pg_statio`; para obter informações adicionais deve ser consultado o Capítulo 23 .

`stats_row_level (boolean)`

Habilita a coleta de estatísticas no nível de linha da atividade do banco de dados. O valor padrão é desabilitado. Se estiver habilitado, os dados produzidos podem ser acessados através da família de visões do sistema `pg_stat` e `pg_statio`; para obter informações adicionais deve ser consultado o Capítulo 23 .

`stats_reset_on_server_start` (boolean)

Se estiver habilitado, as estatísticas coletadas são zeradas sempre que o servidor é reinicializado. Se estiver desabilitado, as estatísticas são acumuladas entre as reinicializações do servidor. O valor padrão é habilitado. Somente pode ser definido na inicialização do servidor.

## 16.4.8. Padrões de conexão do cliente

### 16.4.8.1. Comportamento da declaração

`search_path` (string)

Esta variável especifica a ordem de procura nos esquemas quando um objeto (tabela, tipo de dado, função, etc.) é referenciado simplesmente por um nome, sem o componente do esquema. Quando existem objetos com nomes idênticos em esquemas diferentes, é utilizado o que for encontrado primeiro no caminho de procura. Um objeto que não está em nenhum dos esquemas do caminho de procura, somente pode ser referenciado especificando o esquema que o contém usando um nome qualificado (com ponto).

O valor de `search_path` deve ser uma lista de nomes de esquemas separados por vírgula. Se um dos itens da lista for o valor especial `$user`, então este valor é substituído pelo esquema que tem o nome retornado por `SESSION_USER`, caso este esquema exista (caso contrário, `$user` é ignorado).

É sempre feita a procura no esquema do catálogo do sistema, `pg_catalog`, esteja presente no caminho de procura ou não. Se estiver mencionado no caminho de procura, então a procura será feita na ordem especificada. Se `pg_catalog` não estiver presente no caminho de procura, então será feita a procura neste esquema *antes* de ser feita a procura em qualquer um dos esquemas do caminho de procura. Também deve ser observado que é feita a procura no esquema de tabela temporária, `pg_temp_nnn`, antes de ser feita em qualquer um dos outros esquemas.

Quando os objetos são criados sem especificar o esquema de destino, são colocados no primeiro esquema da lista do caminho de procura. Quando o caminho de procura está vazio é gerado um erro.

O valor padrão para este parâmetro é '`$user, public`' (onde a segunda parte é ignorada quando não há um esquema chamado `public`). Dá suporte ao uso compartilhado de um banco de dados (onde nenhum usuário possui um esquema privativo, e todos compartilham o esquema `public`), esquemas privativos por usuário, e a combinação destes. Podem ser obtidos outros resultados mudando a definição do caminho de procura padrão, tanto globalmente quanto por usuário.

O valor efetivo corrente do caminho de procura pode ser examinado através da função SQL `current_schemas()`. Não é exatamente o mesmo que examinar o valor de `search_path`, uma vez que `current_schemas()` mostra como as solicitações que aparecem em `search_path` foram resolvidas. Para obter informações adicionais sobre a função `current_schemas()` deve ser consultada a Tabela 9-41.

Para obter informações adicionais sobre manuseio de esquemas deve ser consultada a Seção 5.8.

`default_tablespace` (string)

Esta variável especifica o espaço de tabelas onde serão criados os objetos (tabelas e índices), quando o comando `CREATE` não especificar explicitamente o espaço de tabelas.

O valor pode ser o nome de um espaço de tabelas, ou uma cadeia de caracteres vazia para especificar o espaço de tabelas padrão do banco de dados corrente. Se o valor não corresponder a um espaço de tabelas existente, o PostgreSQL utiliza automaticamente o espaço de tabelas padrão do banco de dados corrente.

Para obter informações adicionais sobre espaços de tabelas deve ser consultada a Seção 18.6.

`check_function_bodies` (boolean)

Quando é definido como `false`, desabilita a validação da cadeia de caracteres do corpo da função durante a execução do comando `CREATE FUNCTION`. Normalmente definido como `true`. Ocasionalmente é útil desabilitar a validação para evitar problemas como referências à frente ao restaurar as definições das funções a partir de cópia de segurança.

`default_transaction_isolation` (string)

Cada transação SQL possui um nível de isolamento, que pode ser “READ UNCOMMITTED”, “READ COMMITTED”, “REPEATABLE READ” ou “SERIALIZABLE”. Este parâmetro controla o nível de isolamento padrão de cada nova transação. O valor padrão é “READ COMMITTED”.

Para obter informações adicionais devem ser consultados o Capítulo 12 e SET TRANSACTION

`default_transaction_read_only` (boolean)

Uma transação SQL apenas de leitura não pode alterar tabelas que não sejam temporárias. Este parâmetro controla o status de apenas para leitura padrão para cada nova transação. O valor padrão é `false` (leitura/escrita).

Para obter informações adicionais deve ser consultado SET TRANSACTION.

`statement_timeout` (integer)

Interrompe qualquer declaração que exceda o número de milissegundos especificado. O valor zero, que é o valor padrão, desabilita esta limitação.

#### 16.4.8.2. Idioma e formatação

`DateStyle` (string)

Define o formato de exibição para os valores de data e hora, assim como as regras para interpretar valores de entrada de data ambíguos. Por motivos históricos esta variável contém dois componentes independentes: a especificação do formato de saída (ISO, Postgres, SQL ou German), e a especificação para entrada/saída da ordem de dia, mês e ano na data (DMY, MDY, ou YMD). Podem ser definidos separadamente ou juntos. As palavras chave `Euro` e `European` são sinônimos para `DMY`; as palavras chave `US`, `NonEuro`, e `NonEuropean` são sinônimos para `MDY`. Para obter informações adicionais deve ser consultada a Seção 8.5. O valor padrão é `ISO, MDY`.

`timezone` (string)

Define a zona horária para exibir e interpretar os carimbos do tempo. O valor padrão é 'unknown', o que significa utilizar o que estiver especificado no ambiente do sistema operacional para zona horária. Para obter informações adicionais deve ser consultada a Seção 8.5.

`australian_timezones` (boolean)

Se estiver definido como verdade, ACST, CST, EST e SAT são interpretados como zonas horárias da Austrália, em vez de zonas horárias das Américas e Sábado. O valor padrão é falso.

`extra_float_digits` (integer)

Ajusta o número de dígitos mostrados em valores de ponto flutuante, incluindo `float4`, `float8` e tipos de dado geométricos. O valor do parâmetro é adicionado ao número de dígitos padrão (`FLT_DIG` e `DBL_DIG`, conforme seja apropriado). O valor pode ser definido tão alto quanto 2, para incluir dígitos parcialmente significativos; é especialmente útil para dados de ponto flutuante em cópias de segurança, que precisem ser restaurados com exatidão. Também pode ser definido com um valor negativo para suprimir dígitos não desejados.

`client_encoding` (string)

Define a codificação (conjunto de caracteres) do lado cliente. O padrão é utilizar a codificação do banco de dados.

`lc_messages` (string)

Define a linguagem em que as mensagens são mostradas. Os valores aceitos são dependentes do sistema; para obter informações adicionais deve ser consultada a Seção 20.1. Se esta variável for definida como uma cadeia de caracteres vazia (que é o padrão), então o valor é herdado do ambiente de execução do servidor de uma maneira dependente do sistema.

Em alguns sistemas esta categoria de idioma não existe; definir esta variável funciona, mas não produz nenhum efeito. Existe, também, a chance de não haver mensagens traduzidas para o idioma desejado; neste caso, vão continuar sendo vistas mensagens em Inglês.

`lc_monetary` (string)

Define o idioma a ser utilizado para formatar quantias monetárias como, por exemplo, na família de funções `to_char`. Os valores aceitos são dependentes do sistema; para obter mais informações deve ser consultada a Seção 20.1. Se esta variável for definida como uma cadeia de caracteres vazia (que é o padrão), então o valor é herdado do ambiente de execução do servidor de uma maneira dependente do sistema.

`lc_numeric(string)`

Define o idioma a ser utilizado para formatar números como, por exemplo, na família de funções `to_char()`. Os valores aceitos são dependentes do sistema; para obter mais informações deve ser consultada a Seção 20.1. Se esta variável for definida como uma cadeia de caracteres vazia (que é o padrão), então o valor é herdado do ambiente de execução do servidor de uma maneira dependente do sistema.

`lc_time(string)`

Define o idioma a ser utilizado para formatar valores de data e hora (Atualmente esta definição não faz nada, mas poderá fazer no futuro). Os valores aceitos são dependentes do sistema; para obter mais informações deve ser consultada a Seção 20.1. Se esta variável for definida como uma cadeia de caracteres vazia (que é o padrão), então o valor é herdado do ambiente de execução do servidor de uma maneira dependente do sistema.

### 16.4.8.3. Outros padrões

`explain_pretty_print(boolean)`

Determina se `EXPLAIN VERBOSE` utiliza a formatação com recuos ou sem recuos para mostrar o conteúdo detalhado das árvores de comando. O valor padrão é habilitado.

`dynamic_library_path(string)`

Se for necessário abrir um módulo carregável dinamicamente, sem que o nome de arquivo especificado no comando `CREATE FUNCTION` ou no comando `LOAD` possua componente de diretório (ou seja, o nome não contém o caractere barra), o sistema procura pelo arquivo requisitado usando este caminho.

O valor de `dynamic_library_path` deve ser uma lista de caminhos absolutos de diretório separados por dois-pontos. Se o elemento da lista começar pela cadeia de caracteres especial `$libdir`, este valor é substituído pelo diretório de biblioteca do pacote PostgreSQL compilado. Este é o local onde os módulos fornecidos na distribuição do PostgreSQL são instalados (deve ser utilizado o comando `pg_config --pkglibdir` para descobrir o nome deste diretório<sup>9</sup>). Por exemplo:

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/my_project/lib:$libdir'
```

ou, no ambiente Windows:

```
dynamic_library_path = 'C:\tools\postgresql;H:\my_project\lib;$libdir'
```

O valor padrão deste parâmetro é `'$libdir'`. Se o valor for definido como uma cadeia de caracteres vazia, é desabilitada a procura automática no caminho.

Este parâmetro pode ser modificado em tempo de execução pelos superusuários, mas a definição feita desta maneira somente é preservada até o final da conexão do cliente, portanto este método deve ser reservado para fins de desenvolvimento. A maneira recomendada para definir este parâmetro é através do arquivo de configuração `postgresql.conf`.

### 16.4.9. Gerenciamento de bloqueio

`deadlock_timeout(integer)`

Quantidade de tempo, em milissegundos, para aguardar pelo término do bloqueio antes de verificar a existência de uma condição de impasse (`deadlock`). A verificação de impasse é relativamente lenta e, portanto, o servidor não faz esta verificação toda vez que aguarda pelo término do bloqueio. Nós (otimistas?) assumimos que os bloqueios não são comuns em aplicativos em produção e, simplesmente, aguardamos pelo término do bloqueio por algum tempo antes de começar a verificar a situação de impasse. O aumento deste valor reduz a quantidade de tempo desperdiçada em verificações de impasse desnecessárias, mas atrasa o relato de erros de impasse verdadeiros. O valor padrão é 1000 (ou seja, um segundo), que provavelmente é o menor valor desejado na prática. Em servidores muito carregados pode ser desejado aumentar este valor. A definição ideal deve ser um valor superior ao tempo típico de uma transação, para melhorar a chance do bloqueio ser liberado antes de decidir verificar o impasse.

`max_locks_per_transaction(integer)`

A tabela de bloqueio compartilhada é dimensionada pela hipótese de que é necessário bloquear, no máximo, `max_locks_per_transaction * max_connections` objetos distintos de uma vez (Desta forma, o nome deste parâmetro pode confundir: não é um limite rígido do número de bloqueios de alguma transação, mas sim o valor

médio máximo). O valor padrão, 64, tem se mostrado historicamente suficiente, mas pode ser necessário aumentar este valor quando há clientes que acessam muitas tabelas diferentes em uma única transação. Somente pode ser definido na inicialização do servidor.

## 16.4.10. Compatibilidade de versão e de plataforma

### 16.4.10.1. Versões anteriores do PostgreSQL

`add_missing_from` (boolean)

Quando o valor for igual a `true`, as tabelas referenciadas por uma consulta serão automaticamente adicionadas à cláusula `FROM`, caso não estejam presentes. O valor padrão é `true` para manter a compatibilidade com as versões anteriores do PostgreSQL. Entretanto, este comportamento não é SQL-padrão, e muitas pessoas não gostam porque pode esconder enganos. Deve ser definido como `false` para obter o comportamento SQL-padrão de rejeitar referências a tabelas que não estão presentes na cláusula `FROM`.

`regex_flavor` (string)

A “variedade” (`flavor`>) de expressão regular pode ser definida como `advanced`, `extended` ou `basic`. O valor padrão é `advanced`. Definir como `extended` pode ser útil para manter a compatibilidade exata com as versões do PostgreSQL anteriores a 7.4. Para obter detalhes deve ser consultada a Seção 9.7.3.1.

`sql_inheritance` (boolean)

Controla a semântica da herança, em particular se as subtabelas são incluídas por padrão em vários comandos. Não eram incluídas nas versões anteriores a 7.1. Se for necessário usar o comportamento antigo esta variável pode ser definida como desabilitada, mas a longo prazo incentiva-se mudar os aplicativos para que passem a utilizar a palavra chave `ONLY` para excluir as subtabelas. Para obter informações adicionais sobre herança deve ser consultada a Seção 5.5.

`default_with_oids` (boolean)

Controla se os comandos `CREATE TABLE` e `CREATE TABLE AS` incluem a coluna `OID` nas tabelas criadas, caso não seja especificado `WITH OIDS` nem `WITHOUT OIDS`. Também determina se os `OIDs` serão incluídos nas tabelas criadas através do comando `SELECT INTO`. No PostgreSQL 8.0.0 o valor padrão para `default_with_oids` é habilitado. Este é também o comportamento das versões anteriores do PostgreSQL. Entretanto, não se encoraja que seja assumido que as tabelas irão conter `OIDs` por padrão. Provavelmente o valor padrão desta opção será igual a desabilitado nas versões futuras do PostgreSQL.

Para facilitar a compatibilidade com os aplicativos que utilizam `OIDs`, esta opção deve ser deixada habilitada. Para facilitar a compatibilidade com as versões futuras do PostgreSQL, esta opção deve ser desabilitada, e os aplicativos que necessitarem de `OIDs` para determinadas tabelas devem especificar, explicitamente, `WITH OIDS` no momento de criação das tabelas.

### 16.4.10.2. Compatibilidade de plataforma e cliente

`transform_null_equals` (boolean)

Quando habilitado, as expressões da forma `expr = NULL` (ou `NULL = expr`) são tratadas como `expr IS NULL`, ou seja, retornam verdade se `expr` resultar em um valor nulo, e falso caso contrário. O comportamento correto de `expr = NULL` é sempre retornar nulo (desconhecido). Portanto, o valor padrão desta opção é desabilitado.

Entretanto, no Microsoft Access formulários filtrados produzem consultas que parecem utilizar `expr = NULL` para testar valores nulos e, portanto, se for utilizada esta interface para acessar o banco de dados pode ser necessário habilitar esta opção. Uma vez que as expressões da forma `expr = NULL` sempre retornam o valor nulo (utilizando a interpretação correta), não são muito úteis e, por isso, não aparecem com frequência nos aplicativos normais, portanto esta opção produz pouco dano na prática. Porém, os novos usuários ficam frequentemente confusos sobre a semântica das expressões que envolvem valores nulos e, portanto, esta opção não é habilitada por padrão.

Deve ser observado que esta opção afeta apenas a forma exata `= NULL`, e não os outros operadores de comparação ou outras expressões computacionalmente equivalentes a alguma expressão envolvendo o operador de igual (como o `IN`). Portanto, esta opção não é uma solução geral para má programação.

Para ver informações relacionadas deve ser consultada a Seção 9.2 .

### 16.4.11. Opções pré-definidas

Os “parâmetros” que se seguem são apenas para leitura, sendo determinados quando o PostgreSQL é compilado, ou quando é instalado. Desta forma, foram excluídos do arquivo `postgresql.conf` modelo. Estes parâmetros dão informações sobre vários aspectos do comportamento do PostgreSQL que podem ser de interesse de determinados aplicativos, em particular de aplicativos cliente administrativos.

`block_size (integer)`

Mostra o tamanho de um bloco de disco. É determinado pelo valor de `BLCKSZ` quando o servidor é construído. O valor padrão é 8192 bytes. O significado de algumas variáveis de configuração (como `shared_buffers`) é influenciado por `block_size`. Para obter informações adicionais deve ser consultada a Seção 16.4.3.

`integer_datetimes (boolean)`

Mostra se o PostgreSQL foi construído com suporte para data e hora com inteiros de 64 bits. É definido configurando com `--enable-integer-datetimes` quando o PostgreSQL é construído. O valor padrão é `off`.

`lc_collate (string)`

Mostra o idioma usado para fazer a classificação de dados textuais. Para obter informações adicionais deve ser consultada a Seção 20.1. O valor é determinado quando o agrupamento de bancos de dados é inicializado.

`lc_ctype (string)`

Mostra o idioma que determina a classificação dos caracteres. Para obter informações adicionais deve ser consultada a Seção 20.1. O valor é determinado quando o agrupamento de bancos de dados é inicializado. Normalmente é a mesma utilizada em `lc_collate`, mas pode ser definido um valor diferente para aplicações especiais.

`max_function_args (integer)`

Mostra o número máximo de argumentos de uma função. É determinado pelo valor de `FUNC_MAX_ARGS` quando o servidor é construído. O valor padrão é 32.

`max_identifier_length (integer)`

Mostra o comprimento máximo do identificador. É determinado como sendo o valor de `NAMEDATALEN` menos 1 quando o servidor é construído. O valor padrão de `NAMEDATALEN` é 64; portanto o valor padrão de `max_identifier_length` é 63.

`max_index_keys (integer)`

Mostra o número máximo de chaves de índice. É determinado pelo valor de `INDEX_MAX_KEYS` quando o servidor é construído. O valor padrão é 32.

`server_encoding (string)`

Mostra a codificação do banco de dados (conjunto de caracteres). É determinado quando o banco de dados é criado. Normalmente os clientes precisam se preocupar apenas com o valor de `client_encoding`.

`server_version (string)`

Mostra o número da versão do servidor. É determinado pelo valor de `PG_VERSION` quando o servidor é construído.

### 16.4.12. Opções personalizadas

Esta funcionalidade foi projetada para permitir que os módulos complementares (como as linguagens procedurais) adicionem opções que normalmente não são conhecidas pelo PostgreSQL. Isto permite que os módulos complementares sejam configurados da maneira padrão.

`custom_variable_classes (string)`

Especifica um ou vários nomes de classe a serem utilizados pelas variáveis personalizadas, na forma de uma lista separada por vírgulas. Uma variável personalizada é uma variável que normalmente não é conhecida pelo próprio PostgreSQL, mas que é utilizada por algum módulo complementar. Estas variáveis devem possuir nomes formados pelo nome da classe, um ponto, e o nome da variável. O parâmetro `custom_variable_classes` especifica todos os nomes de classe utilizados por uma determinada instalação. Somente pode ser definido na inicialização do servidor, ou no arquivo `postgresql.conf`.



A dificuldade em definir variáveis personalizadas no arquivo `postgresql.conf`, é que este arquivo deve ser lido antes dos módulos complementares serem carregados e, portanto, normalmente as variáveis personalizadas são rejeitadas por serem desconhecidas. Quando é definido o parâmetro `custom_variable_classes`, o servidor aceita definições de variáveis arbitrárias para as classes especificadas. Estas variáveis são tratadas como posicionadores (`placeholders`), não tendo nenhuma função até que o o módulo que as define seja carregado. Quando o módulo de uma determinada classe é carregado, este adiciona as definições apropriadas de variáveis para seu nome de classe, converte os valores dos posicionadores de acordo com estas definições, e emite advertências para todos os posicionadores de sua classe remanescentes (que se presume como sendo variáveis de configuração escritas de forma errada).

Abaixo está mostrado um exemplo do que o arquivo `postgresql.conf` deve conter quando são usadas variáveis personalizadas:

```
custom_variable_classes = 'plr,pljava'
plr.path = '/usr/lib/R'
pljava.foo = 1
plruby.bar = true          # gera erro, nome de classe desconhecido
```

### 16.4.13. Opções do desenvolvedor

As opções a seguir foram feitas para funcionar no código fonte do PostgreSQL e, em alguns casos, ajudar na recuperação de bancos de dados seriamente danificados. Não deve haver razão para usá-las na configuração de um banco de dados de produção. Por isso, foram excluídas do arquivo `postgresql.conf` modelo. Deve ser observado que muitas destas opções requerem sinalizadores especiais na compilação do código fonte para que funcionem.

`debug_assertions (boolean)`

Habilita várias verificações de asserção. É uma ajuda de depuração. Se estiverem acontecendo problemas estranhos, ou quedas, pode se querer habilitá-la, porque podem ser mostrados erros de programação. Para utilizar esta opção a macro `USE_ASSERT_CHECKING` deverá estar definida quando o PostgreSQL for construído (realizado pela opção `--enable-cassert` do `configure`). Deve ser observado que `debug_assertions` é habilitado por padrão quando o PostgreSQL é construído com as asserções habilitadas.

`debug_shared_buffers (integer)`

Número de segundos entre os relatos de lista de `buffers` livres. Se for maior que zero emite estatísticas `buffers` livres para o registro ao intervalo desta quantidade de segundos. O valor padrão igual a zero desabilita o relato.

`pre_auth_delay (integer)`

Se for diferente de zero, logo após um novo processo servidor ser lançado (`forked`) ocorre um retardo desta quantidade de segundos, antes de realizar o processo de autenticação. Tem por finalidade dar oportunidade de anexar o processo servidor a um depurador para acompanhar um mal comportamento na autenticação.

`trace_notify (boolean)`

Gera uma grande quantidade de saída de depuração para os comandos `LISTEN` e `NOTIFY`. O valor do parâmetro `client_min_messages` ou `log_min_messages` deve ser `DEBUG1`, ou inferior, para que esta saída seja enviada para o `log` do cliente ou do servidor, respectivamente.

`trace_locks (boolean)`

`trace_lwlocks (boolean)`

`trace_userlocks (boolean)`

`trace_lock_oidmin (boolean)`

`trace_lock_table (boolean)`

`debug_deadlocks (boolean)`

`log_btree_build_stats (boolean)`

Várias outras opções de rastreamento e depuração de código.

`wal_debug (boolean)`

Se o valor for verdade, emite saída de depuração relacionada com o WAL. Esta opção somente estará disponível se a macro `WAL_DEBUG` for definida quando o PostgreSQL for compilado.

`zero_damaged_pages` (boolean)

A detecção de um cabeçalho de página danificado normalmente faz com que o PostgreSQL relate um erro e interrompa o comando corrente. Definir `zero_damaged_pages` como verdade faz com que em vez disto o sistema relate uma advertência, limpe a página danificada, e continue o processamento. Este comportamento *destrói os dados*, especificamente todas as linhas na página danificada, mas permite prosseguir após o erro e ler as linhas da tabela porventura presentes em páginas não danificadas. Portanto, é útil para recuperar dados se a corrupção tiver ocorrido devido à erro de máquina ou de software. Geralmente esta opção não deve ser definida como verdade, até que se tenha perdido a esperança de recuperar os dados das páginas danificadas da tabela. A definição padrão é desabilitado, e somente pode ser modificado por um superusuário.

#### 16.4.14. Opções curtas

Para facilitar também estão disponíveis, para alguns parâmetros, chaves de opção de linha de comando de uma única letra, conforme descrito na Tabela 16-1.

**Tabela 16-1. Chave de opção curta**

Opção curta	Equivalente
<code>-B x</code>	<code>shared_buffers = x</code>
<code>-d x</code>	<code>log_min_messages = DEBUGx</code>
<code>-F</code>	<code>fsync = off</code>
<code>-h x</code>	<code>listen_addresses = x</code>
<code>-i</code>	<code>listen_addresses = '*'</code>
<code>-k x</code>	<code>unix_socket_directory = x</code>
<code>-l</code>	<code>ssl = on</code>
<code>-N x</code>	<code>max_connections = x</code>
<code>-p x</code>	<code>port = x</code>
<code>-fi, -fh, -fm, -fn, -fs, -ft<sup>a</sup></code>	<code>enable_indexscan = off, enable_hashjoin = off, enable_mergejoin = off, enable_nestloop = off, enable_seqscan = off, enable_tidscan = off</code>
<code>-s<sup>a</sup></code>	<code>log_statement_stats = on</code>
<code>-S x<sup>a</sup></code>	<code>work_mem = x</code>
<code>-tpa, -tpl, -te<sup>a</sup></code>	<code>log_parser_stats = on, log_planner_stats = on, log_executor_stats = on</code>
Notas: a. Por motivos históricos estas opções devem ser passadas para os processos servidor individuais através da opção <code>-o</code> do <code>postmaster</code> como, por exemplo, <pre>\$ postmaster -o '-S 1024 -s'</pre> ou através da variável de ambiente <code>PGOPTIONS</code> no lado cliente, conforme explicado acima.	

## 16.5. Gerência dos recursos do núcleo

Uma instalação grande do PostgreSQL pode exaurir, rapidamente, vários limites de recurso do sistema operacional (Em alguns sistemas, o padrão de distribuição é tão baixo que, na verdade, nem é necessária uma instalação “grande” para exaurir os recursos). Caso tenha deparado com este tipo de problema, prossiga a leitura.

### 16.5.1. Memória compartilhada e semáforos

A memória compartilhada e os semáforos são referenciados coletivamente como “System V IPC”<sup>10</sup> (junto com as filas de mensagens, que não são relevantes para o PostgreSQL). Quase todo sistema operacional moderno fornece estas

funcionalidades, mas nem todos as têm ativas ou com tamanho suficiente por padrão, especialmente os sistemas herdados do BSD (Para os `ports` para o QNX e para o BeOS, o PostgreSQL fornece sua própria implementação de substituição destas funcionalidades).

A completa falta destas funcionalidades é geralmente manifestada por um erro de “Chamada de sistema ilegal” na inicialização do servidor. Neste caso, não existe nada a ser feito além de reconfigurar o núcleo. O PostgreSQL não funciona sem estas funcionalidades.

Quando o PostgreSQL excede um dos vários limites rígidos do IPC, o servidor se recusa a inicializar, e deve deixar uma mensagem de erro instrutiva descrevendo o problema encontrado e o que fazer sobre o mesmo (Também deve ser consultada a Seção 16.3.1.) Os parâmetros relevantes do núcleo possuem nomes consistentes entre sistemas diferentes; A Tabela 16-2 mostra uma visão geral. Entretanto, os métodos para defini-los variam. Abaixo são fornecidas sugestões para algumas plataformas. Não se deve esquecer que geralmente é necessário reinicializar o computador e até, possivelmente, recompilar o núcleo para alterar estas definições.

**Tabela 16-2. Parâmetros do IPC do System V**

Nome	Descrição	Valores razoáveis
SHMMAX	Tamanho máximo de um segmento de memória compartilhada (bytes) <sup>a</sup>	250 kB + 8.2 kB * <code>shared_buffers</code> + 14.2 kB * <code>max_connections</code> até o infinito
SHMMIN	Tamanho mínimo de um segmento de memória compartilhada (bytes)	1
SHMALL	Quantidade total de memória compartilhada disponível (bytes ou páginas)	se em bytes, o mesmo que SHMMAX; se em páginas, <code>ceil(SHMMAX/PAGE_SIZE)</code>
SHMSEG	Número máximo de segmentos de memória compartilhada por processo	somente 1 segmento é necessário, mas o padrão é muito maior
SHMMNI	Número máximo de segmentos de memória compartilhada para todo o sistema	como SHMSEG mais espaço para outras aplicações
SEMMNI	Número máximo de identificadores de semáforos (ou seja, conjuntos)	pelo menos <code>ceil(max_connections / 16)</code>
SEMMNS	Número máximo de semáforos para todo o sistema	<code>ceil(max_connections / 16) * 17</code> mais espaço para outras aplicações
SEMMSL	Número máximo de semáforos por conjunto	pelo menos 17
SEMAP	Número de entradas no mapa de semáforos <sup>b</sup>	consulte o texto
SEVMX	Valor máximo de um semáforo	pelo menos 1000 (O padrão é geralmente 32767; não mude a não ser quando for obrigado a fazê-lo)
Notas: a. A função <code>shmget()</code> é utilizada para obter acesso a um segmento de memória compartilhada. Esta função falha se o valor do tamanho for menor que SHMMIN ou maior do que SHMMAX. Sun Product Documentation ( <a href="http://docs.sun.com/db/doc/801-6736/6i13fom0j?a=view">http://docs.sun.com/db/doc/801-6736/6i13fom0j?a=view</a> ) (N. do T.) b. SEMMAP deve ser definido como o produto de SEMMNI e SEMMSL: (SEMAP = SEMMNI * SEMMSL). Setting our sights on semaphores ( <a href="http://www.carumba.com/talk/random/swol-10-insidesolaris-2.html">http://www.carumba.com/talk/random/swol-10-insidesolaris-2.html</a> ) (N. do T.)		

O parâmetro de memória compartilhada mais importante é SHMMAX, o tamanho máximo, em bytes, de um segmento de memória compartilhada. Se for recebida uma mensagem de erro da função `shmget` como “Argumento inválido”, é provável que este limite tenha sido excedido. O tamanho do segmento de memória compartilhada requerido varia tanto com o número de `buffers` requisitados (opção `-B`) quanto com o número de conexões permitidas (opção `-N`), embora esta

última seja mais significativa (É possível, como uma solução temporária, diminuir estas definições para eliminar o problema). Como uma aproximação grosseira, o tamanho de segmento requerido pode ser estimado conforme sugerido em Tabela 16-2. Toda mensagem de erro que vier a ser recebida conterá o tamanho da alocação requerida que falhou.

Alguns sistemas também possuem um limite para a quantidade total de memória compartilhada do sistema (`SHMALL`). Deve-se ter certeza que é grande o suficiente para o PostgreSQL mais as outras aplicações que utilizam segmentos de memória compartilhada. (Cuidado: em muitos sistemas `SHMALL` é medido em páginas, e não em bytes).

Menos provável de causar problema é o tamanho mínimo para os segmentos de memória compartilhada (`SHMMIN`), que deve ser no máximo aproximadamente 256 kB para o PostgreSQL (geralmente é apenas 1). O número máximo de segmentos para todo o sistema (`SHMMNI`) ou por processo (`SHMSEG`) não devem causar problema a menos que o sistema os tenha definido como zero.

O PostgreSQL utiliza um semáforo por conexão permitida (opção `-N`), em conjuntos de 16. Cada um destes conjuntos também contém um 17º semáforo contendo um “número mágico”, para detectar colisões com conjuntos de semáforos utilizados por outras aplicações. O número máximo de semáforos no sistema é definido por `SEMMNS` que, conseqüentemente, deve ser pelo menos tão alto quanto `max_connections` mais um adicional para cada 16 conexões permitidas (veja a fórmula na Tabela 16-2). O parâmetro `SEMMNI` determina o limite do número de conjuntos de semáforos que podem existir no sistema de uma vez. Portanto, este parâmetro deve ser pelo menos igual a `ceil(max_connections / 16)`. Diminuir o número de conexões permitidas é um recurso temporário para evitar falhas, geralmente informadas pela mensagem “Nenhum espaço disponível na unidade” emitida pela função `semget`, que confunde.

Em alguns casos também pode ser necessário aumentar `SEMMAP`, para que fique pelo menos na ordem de grandeza de `SEMMNS`. Este parâmetro define o tamanho do mapa de recursos de semáforos, no qual cada bloco contíguo de semáforos disponíveis precisa de uma entrada. Quando um conjunto de semáforos é liberado, este conjunto é adicionado a uma entrada existente adjacente ao bloco liberado, ou é registrado sob uma nova entrada no mapa. Se o mapa estiver cheio, os semáforos liberados serão perdidos (até a reinicialização do servidor). A fragmentação do espaço de semáforos pode, ao longo do tempo, fazer com que haja menos semáforos disponíveis do que deveria haver.

O parâmetro `SEMSL`, que determina quantos semáforos podem existir em um conjunto, deve ser pelo menos igual a 17 para o PostgreSQL.

Várias outras definições relacionadas com “desfazer semáforo”, como `SEMMNU` e `SEMUME`, não são de interesse do PostgreSQL.

## BSD/OS

**Memória compartilhada.** Por padrão, são suportados somente 4 MB de memória compartilhada. Deve-se ter em mente que a memória compartilhada não é paginável; fica bloqueada na RAM. Para aumentar a quantidade de memória compartilhada suportada pelo sistema, deve ser adicionado algo parecido com as linhas mostradas abaixo ao arquivo de configuração do núcleo.

```
options "SHMALL=8192"
options "SHMMAX=\(SHMALL*PAGE_SIZE\)"
```

`SHMALL` é medido em páginas de 4KB, portanto um valor igual a 1024 representa 4 MB de memória compartilhada. Desta forma, as linhas acima aumentam para 32 MB o máximo de área de memória compartilhada. Para os usuários da versão 4.3 ou posterior, provavelmente será necessário aumentar `KERNEL_VIRTUAL_MB` para um valor acima do valor padrão de 248. Uma vez que as alterações tenham sido efetuadas, o núcleo deve ser recompilado e o sistema reinicializado.

Os usuários da versão 4.0, ou anterior, devem utilizar `bpatch` para descobrir o valor de `sysptsize` do núcleo corrente. Este valor é computado dinamicamente durante a inicialização do sistema operacional.

```
$ bpatch -r sysptsize
0x9 = 9
```

Em seguida, deve ser adicionado `SYSPTSIZE` com um valor fixo no arquivo de configuração do núcleo. O valor encontrado utilizando `bpatch` deve ser aumentado; deve ser adicionado 1 para cada 4 MB adicionais de memória compartilhada desejada.

```
options "SYSPTSIZE=16"
```

`sysptsize` não pode ser mudado por `sysctl`.

**Semáforos.** Provavelmente vai ser necessário aumentar o número de semáforos também; o valor total padrão do sistema, igual a 60, permite apenas em torno de 50 conexões do PostgreSQL. Os valores desejados devem ser definidos no arquivo de configuração do núcleo como, por exemplo:

```
options "SEMMNI=40"
options "SEMMNS=240"
```

FreeBSD

NetBSD

OpenBSD

As opções `SYSVSHM` e `SYSVSEM` precisam estar habilitadas quando o núcleo é compilado (Estão por padrão). O tamanho máximo da memória compartilhada é determinado pela opção `SHMMAXPGS` (em páginas). A seguir está mostrado um exemplo de como definir os vários parâmetros:

```
options      SYSVSHM
options      SHMMAXPGS=4096
options      SHMSEG=256

options      SYSVSEM
options      SEMMNI=256
options      SEMMNS=512
options      SEMMNU=256
options      SEMMAP=256
```

(No NetBSD e no OpenBSD a palavra chave é, na verdade, `option`, no singular).

Também pode-se querer configurar o núcleo para que bloqueie a memória compartilhada na RAM, impedindo que seja paginada para a área de troca (`swap`). Deve ser utilizada a definição `kern.ipc.shm_use_phys` do `sysctl`.

HP-UX

A definição padrão costuma ser suficiente para as instalações normais. No HP-UX 10, o padrão de fábrica para `SEMMNS` é 128, que pode ser muito baixo para servidores de banco de dados grandes.

Os parâmetros do IPC podem ser definidos no System Administration Manager (SAM), sob Kernel Configuration→Configurable Parameters. Quando tiver acabado clique em Create A New Kernel.

Linux

O limite padrão de memória compartilhada (tanto `SHMMAX` quanto `SHMALL`) é de 32 MB nos núcleos 2.2, mas pode ser modificado no arquivo de sistema `proc` (sem reinicializar o Linux). Por exemplo, para permitir 128 MB:

```
$ echo 134217728 >/proc/sys/kernel/shmall
$ echo 134217728 >/proc/sys/kernel/shmmax
```

Estes comandos podem ser colocados em um script executado durante a inicialização.

Como alternativa, pode ser utilizado `sysctl`, se estiver disponível, para controlar estes parâmetros. Deve-se procurar pelo arquivo chamado `/etc/sysctl.conf` e adicionar linhas como as mostradas abaixo ao arquivo:

```
kernel.shmall = 134217728
kernel.shmmax = 134217728
```

Este arquivo geralmente é processado durante a inicialização, mas `sysctl` também pode ser chamada explicitamente posteriormente.

Os outros parâmetros possuem tamanho suficiente para qualquer aplicação. Se quiser ver por si próprio, olhe em `/usr/src/linux/include/asm-xxx/shmparam.h` e `/usr/src/linux/include/linux/sem.h`.

MacOS X

No OS X 10.2, e anteriores, deve ser editado o arquivo

`/System/Library/StartupItems/SystemTuning/SystemTuning` e modificados os valores nos seguintes comandos:

```
sysctl -w kern.sysv.shmmax
sysctl -w kern.sysv.shmmin
sysctl -w kern.sysv.shmmni
sysctl -w kern.sysv.shmseg
```

```
sysctl -w kern.sysv.shmall
```

No OS X 10.3 estes comandos foram movidos para `/etc/rc` devendo ser editados neste local. É necessário reinicializar o computador para que as alterações tenham efeito. Deve ser observado que `/etc/rc` geralmente é sobrescrito pelas atualizações do OS X (como 10.3.6 para 10.3.7), portanto é esperado que seja necessário refazer a edição a cada atualização.

Nesta plataforma `SHMALL` é medido em páginas de 4KB.

### SCO OpenServer

Na configuração padrão, somente são permitidos 512 kB de memória compartilhada por segmento, suficiente para cerca de `-B 24 -N 12`. Para aumentar esta definição, primeiro deve-se tornar o diretório `/etc/conf/cf.d` o diretório corrente. Para exibir o valor corrente de `SHMMAX` deve ser executado:

```
./configure -y SHMMAX
```

Para definir um novo valor para `SHMMAX` deve ser executado

```
./configure SHMMAX=valor
```

onde *valor* é o novo valor que se deseja utilizar (em bytes). Após definir `SHMMAX` o núcleo deve ser reconstruído

```
./link_unix
```

e o sistema reinicializado.

### AIX

A partir da versão 5.1, no mínimo, não deve ser necessário fazer qualquer configuração especial para parâmetros como `SHMMAX`, uma vez que parece estar configurado para permitir que toda a memória seja utilizada como memória compartilhada. Este é o tipo de configuração utilizado normalmente para outros bancos de dados como o DB2.

Pode, entretanto, ser necessário modificar a informação global `ulimit` em `/etc/security/limits`, uma vez que os limites rígidos padrão para o tamanho dos arquivos (`fsize`) e número de arquivos (`nofiles`) podem ser muito baixos.

### Solaris

Ao menos na versão 2.6, o tamanho máximo padrão dos segmentos de memória compartilhada é muito baixo para o PostgreSQL. As definições relevantes podem ser mudadas em `/etc/system` como, por exemplo:

```
set shmsys:shminfo_shmmax=0x2000000
set shmsys:shminfo_shmmni=1
set shmsys:shminfo_shmmni=256
set shmsys:shminfo_shmseg=256

set semsys:seminfo_semmmap=256
set semsys:seminfo_semmni=512
set semsys:seminfo_semmns=512
set semsys:seminfo_semmsl=32
```

O computador deve ser reinicializado para as modificações terem efeito.

Também deve ser consultada a página Shared memory uncovered (<http://sunsite.uakom.sk/sunworldonline/swol-09-1997/swol-09-insidesolaris.html>) para obter informações sobre memória compartilhada sob o Solaris.

### UnixWare

No UnixWare 7 o tamanho máximo para os segmentos de memória compartilhada é de 512 kB na configuração padrão. É suficiente para cerca de `-B 24 -N 12`. Para obter o valor corrente de `SHMMAX`, deve ser executado

```
/etc/conf/bin/ldtune -g SHMMAX
```

que mostra os valores corrente, padrão, mínimo e máximo. Para definir um novo valor para `SHMMAX`, deve ser executado

```
/etc/conf/bin/ldtune SHMMAX valor
```

onde *valor* é o novo valor que se deseja utilizar (em bytes). Após definir `SHMMAX`, o núcleo deve ser reconstruído:

```
/etc/conf/bin/ldbuild -B
```

e o sistema reinicializado.

### 16.5.2. Limites de recursos

Os sistemas operacionais da família Unix obrigam respeitar vários tipos de limite de recursos que podem interferir com a operação do servidor PostgreSQL. São de particular importância os limites do número de processos por usuário, de número de arquivos abertos por processo, e a quantidade de memória disponível para cada processo. Cada um destes possui um limite “rígido” e um “flexível”. O limite flexível é o que realmente conta, mas pode ser alterado pelo usuário até o limite rígido. O limite rígido somente pode ser alterado pelo usuário `root`. A chamada de sistema `setrlimit` é responsável pela definição destes parâmetros. São utilizados o comando interno do interpretador de comandos `ulimit` (interpretadores de comandos Bourne) ou `limit` (`csch`) para controlar os limites de recursos a partir da linha de comandos. Nos sistemas derivados do BSD o arquivo `/etc/login.conf` controla os limites dos vários recursos definidos durante a autenticação. Para obter detalhes deve ser vista a documentação do sistema operacional. Os parâmetros relevantes são `maxproc`, `openfiles` e `datasize`. Por exemplo:

```
default:\
...
    :datasize-cur=256M:\
    :maxproc-cur=256:\
    :openfiles-cur=256:\
...
```

(`-cur` é o limite flexível. Deve ser anexado `-max` para definir o limite rígido).

Os núcleos também podem ter limites para alguns recursos para todo o sistema.

- No Linux `/proc/sys/fs/file-max` determina o número máximo de arquivos abertos que o núcleo pode suportar. Pode ser mudado escrevendo um número diferente no arquivo, ou adicionando uma atribuição em `/etc/sysctl.conf`. O limite máximo de arquivos por processo é fixado quando o núcleo é compilado; para obter informações adicionais deve ser consultado o arquivo `/usr/src/linux/Documentation/proc.txt`.

O servidor PostgreSQL utiliza um processo por conexão e, portanto, deve ser especificado pelo menos tantos processos quantas forem as conexões permitidas, em adição ao que for necessário para o restante do sistema. Normalmente não é um problema, mas se forem executados vários servidores na mesma máquina pode ficar apertado.

O limite padrão original para o número de arquivos abertos geralmente é definido como um valor “socialmente amigável”, para permitir a coexistência de muitos usuários em uma mesma máquina sem utilizar uma fração não apropriada dos recursos do sistema. Se forem executados vários servidores na mesma máquina provavelmente é o que se deseja, mas para servidores dedicados pode ser necessário aumentar este limite.

Por outro lado, alguns sistemas podem permitir que cada processo abra um grande número de arquivos; se mais que uns poucos processos o fizerem, então o limite global para todo o sistema pode ser facilmente excedido. Se isto estiver acontecendo, e não for desejado alterar o limite para todo o sistema, pode ser definido o parâmetro de configuração `max_files_per_process` do PostgreSQL para limitar a quantidade de arquivos abertos.

### 16.5.3. Sobre-alocação de memória no Linux

No Linux 2.4 e posteriores, o comportamento padrão de memória virtual não é o ótimo para o PostgreSQL. Devido à maneira como o núcleo implementa a sobre-alocação (`overcommit`) de memória, o núcleo pode fechar o servidor PostgreSQL (o processo `postmaster`), se a demanda por memória de um outro processo fizer com que o sistema fique sem memória virtual.<sup>11</sup>

Caso aconteça, será vista uma mensagem do núcleo parecida com esta (deve ser consultada a documentação e configuração do sistema para achar onde este tipo de mensagem pode ser vista):

```
Out of Memory: Killed process 12345 (postmaster).
```

Esta mensagem indica que o processo `postmaster` foi fechado devido à falta de memória. Embora as conexões com os bancos de dados existentes continuem funcionando normalmente, não é aceita nenhuma nova conexão. Para recuperar é necessário reinicializar o PostgreSQL.

Uma forma de evitar este problema é executar o PostgreSQL em uma máquina onde se tenha certeza que outros processos não vão deixar a máquina sem memória.

No Linux 2.6 e posteriores, uma solução melhor é modificar o comportamento do núcleo para que não haja “sobre-alocação” de memória. Isto é feito selecionando o modo estrito de sobre-alocação através do comando `sysctl`:

```
sysctl -w vm.overcommit_memory=2
```

ou colocando uma entrada equivalente em `/etc/sysctl.conf`. Também pode ser necessário modificar a percentagem de sobre-alocação (`vm.overcommit_ratio`). Para obter detalhes deve ser visto o arquivo de documentação do núcleo `Documentation/vm/overcommit-accounting`.<sup>12</sup>

Existe relato que algumas distribuições do núcleo 2.4 do Linux possuem uma versão inicial do parâmetro `overcommit` do comando `sysctl` da versão 2.6. Entretanto, definir `vm.overcommit_memory` como 2 em um núcleo que não possui o código relevante torna as coisas piores, e não melhores. Recomenda-se a inspeção do código fonte do núcleo utilizado (deve ser consultada a função `vm_enough_memory` no arquivo `mm/mmap.c`), para verificar o que é suportado na versão utilizada antes de tentar utilizar numa instalação 2.4. A presença do arquivo de documentação `overcommit-accounting` não deve ser assumida como uma evidência que a funcionalidade está presente. Em caso de dúvida, deve ser consultado um especialista no núcleo ou o distribuidor do núcleo utilizado.

## 16.6. Parada do servidor

Existem várias formas de parar o servidor de banco de dados. O tipo de parada pode ser controlado através do envio de sinais diferentes para o processo `postmaster`.

### SIGTERM

Após receber o sinal `SIGTERM` o servidor não aceita novas conexões, mas deixa as sessões existentes trabalharem normalmente. A parada é realizada apenas depois de todas as sessões terminarem normalmente. Esta é a *Parada Esperta* (*Smart Shutdown*).

### SIGINT

O servidor não aceita novas conexões e envia para todos os processos servidores existentes o sinal `SIGTERM`, fazendo com que estes interrompam suas transações correntes e terminem imediatamente. Depois aguarda os processos servidor saírem e, finalmente, pára. Esta é a *Parada Rápida* (*Fast Shutdown*).

### SIGQUIT

Esta é a *Parada Imediata* (*Immediate Shutdown*), que faz o processo `postmaster` enviar um sinal `SIGQUIT` para todos os processos descendentes e sair imediatamente, sem parar de forma apropriada. Da mesma maneira, os processos descendentes saem imediatamente após receber o sinal `SIGQUIT`. Provoca uma recuperação (refaz o `log` do WAL) na próxima inicialização. Somente é recomendado em caso de emergência.

O programa `pg_ctl` fornece uma interface conveniente para enviar estes sinais para parar o servidor.

Como alternativa o sinal pode ser enviado diretamente através do comando `kill`. O PID do processo `postmaster` pode ser encontrado utilizando o programa `ps`, ou no arquivo `postmaster.pid` no diretório de dados. Portanto, para efetuar uma parada rápida pode ser utilizado, por exemplo:

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

**Importante:** É melhor não utilizar `SIGKILL` para parar o servidor. Usá-lo impede que o servidor libere a memória compartilhada e os semáforos, o que poderá então ter de ser feito à mão antes de poder inicializar novamente o servidor. Além disso, o sinal `SIGKILL` mata o processo `postmaster` sem deixar que este repasse o sinal para seus subprocessos e, portanto, será necessário matar os subprocessos à mão também.

## 16.7. Conexões TCP/IP seguras com SSL

Para aumentar a segurança criptografando as comunicações cliente/servidor, o PostgreSQL possui suporte nativo para conexões SSL. É necessário que o OpenSSL esteja instalado tanto no cliente quanto no servidor, e que o suporte no PostgreSQL tenha sido habilitado em tempo de construção (consulte o Capítulo 14).



Uma vez compilado com suporte a SSL, o servidor PostgreSQL pode ser inicializado com SSL habilitado, definindo no arquivo `postgresql.conf` o parâmetro `ssl` com o valor `on`. Quando inicializado no modo SSL, o servidor procura pelos arquivos `server.key` e `server.crt` no diretório de dados, que devem conter a chave privada do servidor e o certificado, respectivamente. Estes arquivos devem ser configurados de forma correta para que o servidor possa ser inicializado com SSL habilitado. Se a chave privada estiver protegida por uma frase-senha (`passphrase`), o servidor solicitará esta frase-senha, não inicializando até que seja fornecida.

O servidor atende tanto as conexões comuns quanto as conexões SSL na mesma porta TCP, e negocia com o cliente conectando se vai usar ou não o SSL. Por padrão esta é uma opção do cliente; deve ser visto na Seção 19.1 como configurar o servidor para que seja requerido o uso do SSL para algumas ou para todas as conexões.

Para obter detalhes sobre como criar a chave privada do servidor e o certificado, deve ser consultada a documentação do OpenSSL. Pode ser utilizado para testes um certificado auto-assinado, mas no ambiente de produção deve ser utilizado um certificado assinado por uma autoridade certificadora (CA) (por uma CA global ou por uma local), para que o cliente possa verificar a identidade do servidor. Para criar rapidamente um certificado auto-assinado pode ser utilizado o seguinte comando do OpenSSL:

```
openssl req -new -text -out server.req
```

Devem ser fornecidas as informações solicitadas pelo `openssl`. Assegure-se de especificar “Common Name” como nome do hospedeiro local; a senha desafio (`challenge password`) pode ser deixada em branco. O programa gera uma chave que é protegida por uma frase-senha; não é aceita uma frase-senha com menos de quatro caracteres. Para remover a frase-senha (o que deve ser feito se for desejada uma inicialização automática do servidor), devem ser executados os comandos:

```
openssl rsa -in privkey.pem -out server.key
rm privkey.pem
```

Deve ser fornecida a frase-senha antiga para desbloquear a chave existente. Agora deve ser executado

```
openssl req -x509 -in server.req -text -key server.key -out server.crt
chmod og-rwx server.key
```

para tornar o certificado um certificado auto-assinado, e para copiar a chave e o certificado para onde o servidor procura pelos mesmos.

Se for requerida a verificação dos certificados cliente, devem ser colocados no arquivo `root.crt` no diretório de dados os certificados das CA(s) que se deseja verificar. Quando presente, é requisitado do cliente um certificado cliente durante a inicialização da conexão SSL, que deve ter sido assinado por um dos certificados presentes no arquivo `root.crt`.

Quando o arquivo `root.crt` não está presente, os certificados cliente não são requeridos nem verificados. Neste modo o SSL fornece segurança para a comunicação, mas não para autenticação.

Os arquivos `server.key`, `server.crt` e `root.crt` são examinados apenas durante a inicialização do servidor; o servidor deve ser reinicializado para que as alterações feitas nestes arquivos tenham efeito.

## 16.8. Conexões TCP/IP seguras por túneis SSH

Pode ser utilizado o SSH para criptografar a conexão de rede entre os clientes e o servidor PostgreSQL. Feito de forma apropriada, fornece uma conexão de rede adequadamente segura, mesmo para clientes sem capacidade de SSL.

Primeiro deve-se ter certeza que o servidor SSH está executando de forma apropriada na mesma máquina onde está o servidor PostgreSQL, e que é possível se conectar utilizando o `ssh` como algum usuário. Depois pode ser estabelecido, a partir da máquina cliente, um túnel seguro usando um comando como o mostrado abaixo:

```
ssh -L 3333:foo.com:5432 joel@foo.com
```

O primeiro número no argumento `-L`, 3333, é o número da porta do lado cliente do túnel; pode ser escolhido livremente. O segundo número, 5432, é o fim remoto do túnel: o número da porta que o servidor está utilizando. O nome, ou endereço de IP, entre os números das portas é o hospedeiro do servidor de banco de dados onde vai ser feita a conexão. Para que seja possível conectar com o servidor de banco de dados utilizando este túnel, deve ser feita a conexão com a porta 3333 na máquina cliente:

```
psql -h localhost -p 3333 template1
```

Para o servidor de banco de dados vai parecer como se fosse realmente o usuário `joel@foo.com`, e vai utilizar o procedimento de autenticação que estiver configurado para este usuário e hospedeiro. Deve ser observado que o servidor não vai supor que a conexão está criptografada por SSL, porque, na verdade, não está criptografada entre o servidor SSH e o servidor PostgreSQL. Isto não deve causar nenhum risco adicional à segurança, desde que estes dois servidores estejam na mesma máquina.

Para que a configuração do túnel seja bem-sucedida, deve ser permitida a conexão através do `ssh` como `joel@foo.com`, da mesma maneira como se tivesse tentado utilizar o `ssh` para estabelecer uma sessão de terminal.

**Dica:** Existem diversas outras aplicações que podem fornecer túneis seguros utilizando procedimentos conceitualmente semelhantes ao que foi descrito.

## Notas

1. `daemon` — um programa que em vez de ser chamado explicitamente fica adormecido aguardando acontecer alguma condição. The Jargon File (<http://www.catb.org/~esr/jargon/html/D/daemon.html>) (N. do T.)
2. `ulimit [-SHacdfmnpstuv [limit]]` — Permite controlar os recursos disponíveis para o interpretador de comandos (`shell`), e para os processos iniciados pelo mesmo, nos sistemas que permitem este controle; `-s` especifica o tamanho máximo da pilha (`stack`). (N. do T.)
3. Oracle - Processos em segundo plano - DBWR (Database Writer) — O escritor de banco de dados (DBWR) é responsável pela escrita no disco dos blocos de dados sujos dos `buffers` de blocos de banco de dados. Quando uma transação altera os dados de um bloco de dados, o bloco de dados não precisa ser escrito imediatamente no disco. Portanto, o DBWR pode escrever estes dados no disco de uma maneira mais eficiente que escrever ao término de cada transação. O DBWR geralmente só escreve quando há necessidade de `buffers` de blocos de banco de dados para os dados serem lidos. Os dados são escritos usando o critério “usados menos recentemente”. Oracle Architecture ([http://oracle.basisconsultant.com/oracle\\_architecture.htm](http://oracle.basisconsultant.com/oracle_architecture.htm)) (N. do T.)
4. Consulte `man strftime` — A função `strftime` formata data e hora. (N. do T.)
5. `%a` — O nome abreviado do dia da semana de acordo com o idioma corrente. (N. do T.)
6. `%H` — A hora como número decimal utilizando o relógio de 24 horas (faixa de 00 a 23). (N. do T.)
7. `%M` — O minuto como número decimal (faixa de 00 a 59). (N. do T.)
8. O `syslog`, originalmente escrito por Eric Allman, é um sistema de `log` abrangente. Possui duas funções importantes: liberar os programadores da mecânica entediante de escrever arquivos de `log`, e colocar os administradores controlando o `log`. O formato do arquivo de configuração é “seletor <Tab> ação”, onde o seletor identifica o programa (facilidade) que envia a mensagem de `log` e o nível de severidade da mensagem. Linux Administration Handbook - Evi Nemeth e outros - Prentice Hall PTR. (N. do T.)
9. Também pode ser utilizado o comando `locate plpgsql.so` para ver o nome deste diretório, se `pg_config` não estiver disponível. (N. do T.)
10. Com o System V a AT&T introduziu três novas formas de facilidade de comunicação entre processos (IPC) (filas de mensagens, semáforos e memória compartilhada). Embora o comitê POSIX ainda não tenha completado a padronização destas facilidades, a maioria das implementações dão suporte as mesmas. Além disso, Berkeley (BSD) utiliza soquetes como sua forma primária de IPC, em vez dos elementos do System V. O Linux tem capacidade para utilizar as duas formas de IPC, tanto BSD quanto System V. System V IPC (<http://www.tldp.org/LDP/lpg/node21.html>) (N. do T.)
11. `memory overcommit` — a sobre-alocação de memória é uma funcionalidade do núcleo do Linux que permite as aplicações alocarem mais memória do que realmente existe. A idéia por trás desta funcionalidade é que algumas aplicações alocam grandes quantidades de memória “apenas para o caso de precisarem”, mas na verdade nunca utilizam esta memória. Portanto, a sobre-alocação de memória permite a execução de mais aplicações do que na verdade cabe na memória, desde de que as aplicações realmente não utilizem a memória alocada. Se o fizerem, o núcleo fecha o aplicativo. GNUsound - FAQ (<http://www.gnu.org/software/gnusound/Documentation/ar01s05.html>) (N. do T.)

12. Fedora Core 3 — arquivo `/usr/share/doc/kernel-doc-2.6.11/Documentation/vm/overcommit-accounting` — o núcleo do Linux suporta os seguintes modos de tratamento de sobre-alocação: 0 - tratamento heurístico de sobre-alocação; 1 - sempre faz sobre-alocação; 2 - sem sobre-alocação. (N. do T.)

# Capítulo 17. Usuários do banco de dados e privilégios

Todo agrupamento de bancos de dados possui um conjunto de usuários de banco de dados. Estes usuários são distintos dos usuários gerenciados pelo sistema operacional onde o servidor executa. Os usuários possuem objetos de banco de dados (por exemplo, tabelas), e podem conceder privilégios nestes objetos para outros usuários controlando, assim, quem pode acessar qual objeto.

Este capítulo descreve como criar e gerenciar usuários, e introduz o sistema de privilégios. Mais informações sobre os vários tipos de objetos de banco de dados e os efeitos dos privilégios podem ser encontrados no Capítulo 5.

## 17.1. Usuários de banco de dados

Conceitualmente, os usuários de banco de dados são completamente distintos dos usuários de sistema operacional. Na prática, pode ser conveniente manter correspondência, mas não é requerido. Os nomes dos usuários de banco de dados são globais para todo o agrupamento de bancos de dados (e não próprio de cada banco de dados). Para criar um usuário deve ser utilizado o comando SQL *CREATE USER*:

```
CREATE USER nome_do_usuario;
```

Onde *nome\_do\_usuario* segue as regras dos identificadores SQL: ou não contém caracteres especiais, ou está entre aspas. Para remover um usuário existente deve ser utilizado o comando *DROP USER*:

```
DROP USER nome_do_usuario;
```

Para facilitar, são fornecidos os programas *createuser* e *dropuser* que incorporam estes comandos SQL, e que podem ser executados a partir do interpretador de comandos:

```
createuser nome_do_usuario  
dropuser nome_do_usuario
```

Para conhecer o conjunto de usuários existentes deve ser consultado o catálogo do sistema *pg\_user* como, por exemplo:

```
SELECT username FROM pg_user;
```

Também pode ser utilizado o meta-comando `\du` do programa *psql* para listar os usuários existentes.

Para ser possível ativar o sistema de banco de dados, um sistema recém criado sempre contém um usuário pré-definido. É atribuído o valor 1 para o identificador deste usuário e, por padrão (a menos que seja alterado ao executar o utilitário *initdb*), possui o mesmo nome do usuário de sistema operacional que inicializou o agrupamento de bancos de dados. Geralmente este usuário se chama *postgres*. Para poder criar mais usuários, primeiro é necessário se conectar como este usuário inicial.

Em uma conexão com o servidor de banco de dados, está ativa a identidade de exatamente um usuário. O nome de usuário a ser utilizado em uma determinada conexão com o banco de dados é indicado pelo cliente ao fazer o pedido de conexão, de uma forma específica do aplicativo. Por exemplo, o programa *psql* utiliza a opção `-U` na linha de comando para indicar o usuário a ser utilizado na conexão. Muitos aplicativos assumem, por padrão, o nome do usuário corrente do sistema operacional (inclusive o *createuser* e o *psql*). Portanto, é conveniente manter uma correspondência de nomes entre estes dois conjuntos de usuários.

O conjunto de usuários de banco de dados que podem se conectar através de determinada conexão cliente é determinado pela configuração da autenticação de clientes, conforme explicado no Capítulo 19 (Portanto, um cliente não está necessariamente limitado a se conectar com o mesmo nome de usuário do sistema operacional, da mesma maneira que uma pessoa não está limitada no nome de *login* ao seu nome verdadeiro). Uma vez que a identidade do usuário determina o conjunto de privilégios disponíveis para o cliente conectado, é importante que isto seja definido com cuidado quando se configura um ambiente multiusuário.

## 17.2. Atributos do usuário

O usuário de banco de dados pode possuir vários atributos que definem seus privilégios e interagem com o sistema de autenticação de clientes.

*superuser*

Um superusuário do banco de dados não está sujeito a verificações de permissão. Também, somente um superusuário pode criar novos usuários. Para criar um superusuário do banco de dados deve ser utilizado o comando `CREATE USER nome_do_usuario CREATEUSER`.

*criação de banco de dados*

Para o usuário poder criar bancos de dados deve ser dada uma permissão explícita (exceto para os superusuários, uma vez que estes não estão sujeitos a verificações de permissão). Para criar um usuário que pode criar bancos de dados, deve ser utilizado o comando `CREATE USER nome_do_usuario CREATEDB`.

*senha*

A senha só é importante se o método de autenticação do cliente requerer que o usuário forneça a senha para se conectar ao banco de dados. Os métodos de autenticação `password`, `md5` e `crypt` fazem uso de senha. As senhas de banco de dados são distintas das senhas do sistema operacional. A senha deve ser especificada quando o usuário é criado utilizando `CREATE USER nome_do_usuario PASSWORD 'cadeia de caracteres'`.

Os atributos do usuário podem ser modificados após este ter sido criado utilizando o comando `ALTER USER`. Para obter mais detalhes consulte as páginas de referência dos comandos `CREATE USER` e `ALTER USER`.

O usuário também pode definir padrões pessoais para várias definições de configuração em tempo de execução, conforme descrito na Seção 16.4. Por exemplo, se por alguma razão o usuário desejar desabilitar as varreduras de índice toda vez que se conectar (conselho: não é uma boa idéia), pode utilizar o comando:

```
ALTER USER meu_usuario SET enable_indexscan TO off;
```

Este comando salva a definição (mas não define imediatamente) e nas próximas conexões feitas por este usuário vai parecer que o comando `SET enable_indexscan TO off;` foi chamado logo antes da sessão começar. Continua sendo possível alterar esta definição durante a sessão; apenas será a definição padrão. Para desfazer esta definição deve ser utilizado o comando `ALTER USER meu_usuario RESET nome_da_variável;`

## 17.3. Grupos

Como no Unix, os grupos são uma forma lógica de juntar usuários para facilitar o gerenciamento de privilégios; os privilégios podem ser concedidos, ou revogados, para o grupo como um todo. Para criar um grupo deve ser utilizado o comando SQL `CREATE GROUP`:

```
CREATE GROUP nome_do_grupo;
```

Para adicionar ou remover usuários de um grupo existente deve ser utilizado o comando SQL `ALTER GROUP`:

```
ALTER GROUP nome_do_grupo ADD USER nome_do_usuario, ... ;
ALTER GROUP nome_do_grupo DROP USER nome_do_usuario, ... ;
```

Para remover um grupo deve ser utilizado o comando SQL `DROP GROUP`:

```
DROP GROUP nome_do_grupo;
```

Este comando remove apenas os grupos; não remove os usuários membros do grupo.

Para conhecer o conjunto de grupos existentes, deve ser consultado o catálogo do sistema `pg_group` como, por exemplo:

```
SELECT groname FROM pg_group;
```

Também pode ser utilizado o meta-comando `\dg` do programa `psql` para listar os grupos existentes.

## 17.4. Privilégios

Quando um objeto do banco de dados é criado, é atribuído um dono ao mesmo. O dono é o usuário que executou o comando de criação. Para mudar o dono de uma tabela, índice, sequência ou visão deve ser utilizado o comando *ALTER TABLE*. Por padrão, somente o dono (ou um superusuário) pode fazer qualquer coisa com o objeto. Para permitir o uso por outros usuários, devem ser concedidos *privilégios*.

Existem vários privilégios distintos: *SELECT*, *INSERT*, *UPDATE*, *DELETE*, *RULE*, *REFERENCES*, *TRIGGER*, *CREATE*, *TEMPORARY*, *EXECUTE*, *USAGE* e *ALL PRIVILEGES*. Para obter mais informações sobre os diferentes tipos de privilégio suportados pelo PostgreSQL deve ser consultada a página de referência do comando *GRANT*. O direito de modificar ou remover um objeto é sempre um privilégio apenas de seu dono. É utilizado o comando *GRANT* para conceder privilégios. Portanto, se joel for um usuário existente, e tbl\_contas for uma tabela existente, o privilégio de atualizar a tabela pode ser concedido pelo comando:

```
GRANT UPDATE ON tbl_contas TO joel;
```

Este comando deve ser executado pelo usuário dono da tabela. Para conceder privilégios para um grupo deve ser utilizado o comando:

```
GRANT SELECT ON tbl_contas TO GROUP grp_financas;
```

O nome especial de “usuário” *PUBLIC* pode ser utilizado para conceder o privilégio para todos os usuários do sistema. Escrevendo *ALL* no lugar do privilégio especifica a concessão de todos os privilégios.

Para revogar um privilégio deve ser utilizado o comando *REVOKE*:

```
REVOKE ALL ON tbl_contas FROM PUBLIC;
```

Os privilégios especiais do dono da tabela (ou seja, o direito de *DROP* (remover), *GRANT* (conceder), *REVOKE* (revogar), etc.) são sempre implícitos ao fato de ser o dono, não podendo ser concedidos ou revogados. Mas o dono da tabela pode decidir revogar seus próprios privilégios comuns como, por exemplo, tornando uma tabela somente para leitura para o próprio e para os outros.

## 17.5. Funções e gatilhos

As funções e os gatilhos permitem que usuários insiram código no servidor que outros usuários podem executar sem conhecer. Portanto, estes dois mecanismos permitem a criação de “Cavalos de Tróia”<sup>1</sup> com relativa impunidade. A única proteção real é um controle rígido sobre quem pode definir funções.

As funções executam dentro do processo servidor, com as permissões do sistema operacional do processo servidor de banco de dados. Se a linguagem de programação utilizada pela função permitir acesso à memória sem verificação, é possível mudar as estruturas de dados internas do servidor. Portanto, entre outras coisas, estas funções podem burlar qualquer sistema de controle de acesso. As linguagens de função que permitem este tipo de acesso são consideradas “não confiáveis” (*untrusted*), e o PostgreSQL somente permite que superusuários criem funções escritas nestas linguagens.

## Notas

1. Cavalo de Tróia — Um programa destrutivo disfarçado de um aplicativo benigno. Ao contrário dos vírus, os cavalos de tróia não se replicam, mas podem ser tão destrutivos quanto estes. What is Trojan Horse ([http://www.webopedia.com/TERM/T/Trojan\\_horse.html](http://www.webopedia.com/TERM/T/Trojan_horse.html)). (N. do T.)

# Capítulo 18. Gerenciamento de bancos de dados

Cada instância em execução do servidor PostgreSQL gerencia um ou mais bancos de dados. Os bancos de dados são, portanto, o nível hierárquico mais alto da organização dos objetos SQL (“objetos de banco de dados”). Este capítulo descreve as propriedades dos bancos de dados, e como são criados, gerenciados e removidos.

## 18.1. Visão geral

Um banco de dados é uma coleção nomeada de objetos SQL (“objetos de banco de dados”). Geralmente, todos os objetos de banco de dados (tabelas, funções, etc.) pertencem a um e somente um banco de dados (Mas existem alguns poucos catálogos do sistema como, por exemplo, `pg_database`, que pertencem a todo o agrupamento e podem ser acessados por todos os bancos de dados do agrupamento). Mais precisamente, um banco de dados é uma coleção de esquemas, e os esquemas contêm as tabelas, funções, etc. Portanto, a hierarquia completa é: servidor, banco de dados, esquema, tabela (ou outro tipo de objeto em vez de tabela, como uma função).<sup>1 2 3</sup>

Para se conectar ao servidor de banco de dados, o cliente deve especificar no pedido de conexão o nome do banco de dados que deseja se conectar. Não é possível acessar mais de um banco de dados por conexão (Mas não há restrição quanto ao número de conexões que um aplicativo pode abrir no mesmo ou em outros bancos de dados). Os bancos de dados são fisicamente separados, e o controle de acesso é gerenciado no nível de conexão. Se uma instância do servidor PostgreSQL é usada para abrigar projetos e usuários que devem estar separados e, em sua maioria, desconhecendo um ao outro, é recomendável colocá-los em bancos de dados separados. Se os projetos ou os usuários estão inter-relacionados, devendo um poder utilizar os recursos do outro, devem ser colocados no mesmo banco de dados, mas possivelmente em esquemas separados. Os esquemas são estruturas puramente lógicas, e quem pode acessar o que é gerenciado pelo sistema de privilégios. Podem ser encontradas informações adicionais sobre o gerenciamento de esquemas na Seção 5.8.

Os bancos de dados são criados através do comando `CREATE DATABASE` (consulte a Seção 18.2), e removidos pelo comando `DROP DATABASE` (consulte a Seção 18.5). Para conhecer o conjunto de bancos de dados existentes, deve ser examinado o catálogo do sistema `pg_database` como, por exemplo:

```
SELECT datname FROM pg_database;
```

O meta-comando `\l` e a opção da linha de comando `-l` do aplicativo `psql` são úteis para listar os bancos de dados existentes.

**Nota:** O padrão SQL chama os bancos de dados de “catálogos”, mas na prática não há diferença.

## 18.2. Criação de banco de dados

Para ser possível criar um banco de dados, o servidor PostgreSQL deve estar em execução (consulte a Seção 16.3).

Os bancos de dados são criados através do comando SQL `CREATE DATABASE`:

```
CREATE DATABASE nome_do_banco_de_dados;
```

onde `nome_do_banco_de_dados` segue as regras usuais para identificadores SQL. O usuário corrente se torna, automaticamente, o dono do novo banco de dados. É um privilégio do dono do banco de dados removê-lo posteriormente (o que também remove todos os objetos contidos no banco de dados, mesmo que sejam de outro dono).

A criação de bancos de dados é uma operação restrita. Veja como conceder permissão na Seção 17.2.

Uma vez que é necessário estar conectado ao servidor de banco de dados para poder executar o comando `CREATE DATABASE`, a questão é como pode ser criado o *primeiro* banco de dados de um determinado agrupamento. O primeiro banco de dados é sempre criado pelo utilitário `initdb` quando a área de armazenamento de dados é inicializada (Consulte a Seção 16.2). Este banco de dados se chama `template1`. Portanto, para criar o primeiro banco de dados “de verdade” é necessário se conectar ao banco de dados `template1`.

O nome `template1` (`modelo1`) não é um acidente: Quando se cria um novo banco de dados, o banco de dados modelo é essencialmente clonado. Isto significa que qualquer mudança feita em `template1` é propagada para todos os bancos de

dados criados posteriormente. Implica que o banco de dados modelo não deve ser utilizado para trabalho, mas esta funcionalidade, usada com bom senso, pode ser conveniente. Na Seção 18.3 podem ser vistos mais detalhes.

Para facilitar, também existe um programa que pode ser executado a partir do interpretador de comandos para criar novos bancos de dados, o `createdb`:

```
createdb nome_do_banco_de_dados
```

O programa `createdb` não realiza nenhuma mágica: se conecta ao banco de dados `template1` e executa o comando `CREATE DATABASE`, exatamente como descrito acima. A página de referência de `createdb` contém os detalhes da chamada. Deve ser observado que executar `createdb` sem nenhum argumento cria um banco de dados com o mesmo nome do usuário corrente, que pode ser o desejado, ou não.

**Nota:** O Capítulo 19 contém informações sobre como restringir quem pode se conectar a um determinado banco de dados.

Algumas vezes se deseja criar um banco de dados para outra pessoa. Este usuário deve se tornar o dono do novo banco de dados e, portanto, poder configurá-lo e gerenciá-lo por si próprio. Para fazer isto deve ser utilizado um dos seguintes comandos:

```
CREATE DATABASE nome_do_banco_de_dados OWNER nome_do_usuario;
```

a partir do ambiente SQL, ou

```
createdb -O nome_do_usuario nome_do_banco_de_dados
```

a partir do interpretador de comandos. É necessário ser um superusuário para poder criar bancos de dados para outros usuários.

### 18.3. Bancos de dado modelo

Na verdade o comando `CREATE DATABASE` funciona copiando um banco de dados existente. Por padrão, copia o banco de dados padrão do sistema chamado `template1`. Portanto, este banco de dados é o “modelo” a partir do qual os novos bancos de dados são criados. Se forem adicionados objetos ao `template1`, estes objetos serão copiados nos próximos bancos de dados de usuário criados. Este comportamento permite modificar localmente o conjunto padrão de objetos nos bancos de dados. Por exemplo, se for instalada a linguagem procedural PL/pgSQL em `template1`, esta se tornará automaticamente disponível nos bancos de dados dos usuários sem que precise ser feito qualquer procedimento adicional na criação dos bancos de dados.

Existe um segundo banco de dados padrão do sistema chamado `template0`. Este banco de dados contém os mesmos dados contidos inicialmente em `template1`, ou seja, contém somente os objetos padrão pré-definidos pela versão do PostgreSQL. O banco de dados `template0` nunca deve ser modificado após a execução do utilitário `initdb`. Instruindo o comando `CREATE DATABASE` para copiar `template0` em vez de `template1`, pode ser criado um banco de dados de usuário “intacto”, não contendo nenhuma adição feita ao banco de dados `template1` da instalação local. É particularmente útil ao se restaurar uma cópia de segurança feita por `pg_dump`: o script da cópia de segurança deve ser restaurado em um banco de dados intocado, para garantir a recriação do conteúdo correto da cópia de segurança do banco de dados, sem conflito com as adições que podem estar presentes em `template1`.

Para criar um banco de dados copiando `template0` deve ser utilizado:

```
CREATE DATABASE nome_do_banco_de_dados TEMPLATE template0;
```

a partir do ambiente SQL, ou

```
createdb -T template0 nome_do_banco_de_dados
```

a partir do interpretador de comandos.

É possível criar bancos de dados modelo adicionais e, na verdade, pode ser copiado qualquer banco de dados do agrupamento especificando seu nome como modelo no comando `CREATE DATABASE`. Entretanto, é importante compreender que não há intenção (ainda) que este seja um mecanismo tipo “COPY DATABASE” de uso geral. Em particular, é essencial que o banco de dados de origem esteja inativo (nenhuma transação em andamento alterando dados) durante a



operação de cópia. O comando `CREATE DATABASE` verifica se nenhuma sessão (além da própria) está conectada ao banco de dados de origem no início da operação, mas não garante que não possa haver alteração durante a execução da cópia, resultando em um banco de dados copiado inconsistente. Portanto, recomenda-se que os bancos de dados utilizados como modelo sejam tratados como somente para leitura.

Existem no banco de dados `pg_database` dois sinalizadores úteis para cada banco de dados: as colunas `datistemplate` e `dataallowconn`. A coluna `datistemplate` pode ser definida para indicar que o banco de dados se destina a servir de modelo para o comando `CREATE DATABASE`. Se este sinalizador estiver habilitado, o banco de dados pode ser clonado por qualquer usuário com privilégio de `CREATEDB`; se não estiver habilitado, somente os superusuários e o dono do banco de dados podem cloná-lo. Se `dataallowconn` for falso, então não é permitida nenhuma nova conexão ao banco de dados (mas as sessões existentes não são interrompidas simplesmente definindo o sinalizador como falso). O banco de dados `template0` normalmente é marcado com `dataallowconn = false` para evitar que seja modificado. Tanto `template0` quanto `template1` devem estar sempre marcados com `datistemplate = true`.

Após preparar um banco de dados modelo, ou fazer alguma mudança em um deles, é recomendado executar o comando `VACUUM FREEZE` ou `VACUUM FULL FREEZE` neste banco de dados. Se for feito quando não houver nenhuma outra transação aberta no mesmo banco de dados, é garantido que todas as linhas no banco de dados serão “congeladas” e não estarão sujeitas a problemas de recomeço do ID de transação. Isto é particularmente importante em um banco de dados que terá `dataallowconn` definido como falso, uma vez que não será possível executar a rotina de manutenção `VACUUM` neste banco de dados. Para obter informações adicionais deve ser consultada a Seção 21.1.3.

**Nota:** Os bancos de dados `template1` e `template0` não possuem qualquer status especial além do fato do nome `template1` ser o nome padrão para banco de dados de origem do comando `CREATE DATABASE`, e além de ser o banco de dados padrão para se conectar utilizado por vários programas, como o `createdb`. Por exemplo, `template1` pode ser removido e recriado a partir de `template0` sem qualquer efeito prejudicial. Esta forma de agir pode ser aconselhável se forem adicionadas, por descuido, coisas inúteis ao `template1`.

### Exemplo 18-1. Recriação do banco de dados `template1`

Neste exemplo o banco de dados `template1` é recriado. Deve ser observado na sequência de comandos utilizada que não é possível remover o banco de dados `template1` conectado ao mesmo, e enquanto este banco de dados estiver marcado como modelo no catálogo do sistema `pg_database`.<sup>4</sup>

Para recriar o banco de dados `template1` é necessário: se conectar a outro banco de dados (teste neste exemplo); atualizar o catálogo `pg_database` para que o banco de dados `template1` não fique marcado como um banco de dados modelo; remover e criar o banco de dados `template1`; conectar ao banco de dados `template1`; executar os comandos `VACUUM FULL` e `VACUUM FREEZE`; atualizar o catálogo do sistema `pg_database` para que o banco de dados `template1` volte a ficar marcado como um banco de dados modelo.

Abaixo está mostrada a sequência de comandos utilizada:

```
template1=# DROP DATABASE template1;
ERRO: não é possível remover o banco de dados aberto atualmente
template1=# \c teste
Conectado ao banco de dados "teste".
teste=# DROP DATABASE template1;
ERRO: não é possível remover um banco de dados modelo
teste=# UPDATE pg_database SET datistemplate=false WHERE datname='template1';
UPDATE 1
teste=# DROP DATABASE template1;
DROP DATABASE
teste=# CREATE DATABASE template1 TEMPLATE template0 ENCODING 'latin1';
CREATE DATABASE
teste=# \c template1
Conectado ao banco de dados "template1".
template1=# VACUUM FULL;
VACUUM
template1=# VACUUM FREEZE;
VACUUM
template1=# UPDATE pg_database SET datistemplate=true WHERE datname='template1';
UPDATE 1
```

## 18.4. Configuração do banco de dados

Como foi visto na Seção 16.4, o servidor PostgreSQL possui um grande número de variáveis de configuração em tempo de execução. Para muitas destas variáveis podem ser definidos valores padrão específicos para cada banco de dados.

Por exemplo, se por algum motivo for desejado desabilitar o otimizador GEQO para um determinado banco de dados, normalmente seria necessário desabilitá-lo para todos os bancos de dados, ou ter certeza que cada cliente ao se conectar a este banco de dados vai executar `SET geqo TO off;`. Para tornar esta definição a definição padrão, pode ser executado o comando:

```
ALTER DATABASE meu_banco_de_dados SET geqo TO off;
```

Este comando salva a definição (mas não a define imediatamente), e nas próximas conexões a este banco de dados vai parecer que `SET geqo TO off;` foi executado logo após a sessão iniciar. Deve ser observado que os usuários continuarão podendo alterar esta definição durante a sessão; apenas será a definição padrão. Para desfazer esta definição deve ser utilizado `ALTER DATABASE nome_do_banco_de_dados RESET nome_da_variável;`.

## 18.5. Remoção do banco de dados

Os bancos de dados são removidos através do comando `DROP DATABASE`:

```
DROP DATABASE nome_do_banco_de_dados;
```

Somente o dono do banco de dados (ou seja, o usuário que o criou) ou um superusuário podem remover um banco de dados. A remoção do banco de dados remove todos os objetos contidos no banco de dados. A remoção do banco de dados não pode ser desfeita.

Não é possível executar o comando `DROP DATABASE` estando conectado ao banco de dados a ser removido. É possível, entretanto, estar conectado a qualquer outro banco de dados, inclusive o banco de dados `template1`. O banco de dados `template1` é a única opção para remover o último banco de dados de usuário de um determinado agrupamento.

Para facilitar, também existe um programa que pode ser executado a partir do interpretador de comandos para remover bancos de dados, o `dropdb`:

```
dropdb nome_do_banco_de_dados
```

(Diferentemente do `createdb`, a ação padrão não é remover o banco de dados que tem o mesmo nome do usuário corrente).

## 18.6. Espaços de tabelas

No PostgreSQL os espaços de tabelas permitem aos administradores definir locais no sistema de arquivos onde os arquivos que representam objetos do banco de dados podem ser armazenados. Uma vez criado, o espaço de tabelas pode ser referenciado por seu nome ao criar os objetos do banco de dados.<sup>5 6 7</sup>

Utilizando espaços de tabelas, o administrador pode controlar a organização em disco da instalação do PostgreSQL. É útil pelo menos de duas maneiras:

Primeira: se a partição ou volume onde o agrupamento foi inicializado ficar sem espaço, e não puder ser estendido, pode ser criado um espaço de tabelas em uma partição diferente e utilizado até que o sistema possa ser reconfigurado.

Segunda: os espaços de tabelas permitem que o administrador utilize seu conhecimento do padrão de utilização dos objetos de banco de dados para otimizar o desempenho. Por exemplo, um índice muito utilizado pode ser colocado em um disco muito rápido com alta disponibilidade, como uma unidade de estado sólido.<sup>8</sup> Ao mesmo tempo, uma tabela armazenando dados históricos raramente utilizados, ou que seu desempenho não seja crítico, pode ser armazenada em um sistema de disco mais barato e mais lento.

Para definir um espaço de tabelas é utilizado o comando `CREATE TABLESPACE` como, por exemplo:

```
CREATE TABLESPACE area_veloz LOCATION '/mnt/sda1/postgresql/data';
```

O local deve ser um diretório existente, vazio, pertencente ao usuário de sistema do PostgreSQL. Depois disso, todos os objetos criados neste espaço de tabelas serão armazenados em arquivos sob este diretório.

**Nota:** Geralmente não faz muito sentido criar mais de um espaço de tabelas por sistema de arquivos lógico, uma vez que não se pode controlar o local de cada arquivo dentro do sistema de arquivos lógico. Entretanto, o PostgreSQL não impõe este tipo de restrição e, na verdade, não está preocupado com as fronteiras do sistema de arquivos. Apenas armazena os arquivos nos diretórios onde se informa que devam ser utilizados.

A criação do espaço de tabelas deve ser feito por um superusuário do banco de dados, mas após ser criado pode ser permitido o seu uso pelos usuários comuns. Para isso ser feito deve ser concedido o privilégio `CREATE` para o mesmo.

Podem ser direcionadas tabelas, índices, e bancos de dados inteiros para um determinado espaço de tabelas. Para que isto seja feito, um usuário que possua o privilégio `CREATE` para o espaço de tabelas deve informar o nome do espaço de tabelas no respectivo comando. Por exemplo, o comando abaixo cria uma tabela no espaço de tabelas `espacol`:

```
CREATE TABLE foo(i int) TABLESPACE espacol;
```

Como alternativa, pode ser utilizado o parâmetro `default_tablespace`:

```
SET default_tablespace = espacol;
CREATE TABLE foo(i int);
```

Quando `default_tablespace` é definido como qualquer coisa que não seja uma cadeia de caracteres vazia, este fornece uma cláusula `TABLESPACE` implícita para os comandos `CREATE TABLE` e `CREATE INDEX` que não possuem uma cláusula explícita.

O espaço de tabelas associado com o banco de dados é utilizado para armazenar os catálogos do sistema deste banco de dados, assim como todos os arquivos temporários criados pelos processos servidor que utilizam este banco de dados. Além disso, é o espaço de tabelas padrão usado para as tabelas e índices criados no banco de dados, se a cláusula `TABLESPACE` não for fornecida (explicitamente ou através de `default_tablespace`) quando os objetos são criados. Se o banco de dados for criado sem que seja especificado um espaço de tabelas para o mesmo, é utilizado o mesmo espaço de tabelas do banco de dados modelo do qual é copiado.

São criados, automaticamente, dois espaços de tabelas pelo utilitário `initdb`. O espaço de tabelas `pg_global` é utilizado para os catálogos do sistema compartilhados. O espaço de tabelas `pg_default` é o espaço de tabelas padrão dos bancos de dados `template1` e `template0` (e, portanto, será o espaço de tabelas padrão para todos os outros bancos de dados, a menos que seja mudado pela cláusula `TABLESPACE` do comando `CREATE DATABASE`).

Uma vez criado, o espaço de tabelas pode ser utilizado por qualquer banco de dados, desde que o usuário solicitante tenha os privilégios necessários. Isto significa que o espaço de tabelas não pode ser removido até que todos os objetos de todos os bancos de dados que utilizam o espaço de tabelas sejam removidos.

Para remover um espaço de tabelas vazio deve ser utilizado o comando `DROP TABLESPACE`.

Para conhecer o conjunto de espaços de tabelas existente deve ser consultado o catálogo do sistema `pg_tablespace` como, por exemplo:

```
SELECT spcname FROM pg_tablespace;
```

O meta-comando `\db` do programa `psql` também pode ser utilizado para listar os espaços de tabela existentes.

O PostgreSQL faz amplo uso de vínculos simbólicos para simplificar a implementação de espaços de tabelas. Isto significa que os espaços de tabelas *somente* podem ser utilizados nos sistemas que suportam vínculos simbólicos.<sup>9</sup>

O diretório `$PGDATA/pg_tblspc` contém vínculos simbólicos que apontam para cada um dos espaços de tabela não-nativos definidos no agrupamento. Embora não seja recomendado, é possível mudar a disposição manualmente. Duas advertências: não faça isso com o `postmaster` executando; após reiniciar o `postmaster`, deve ser atualizado o catálogo do sistema `pg_tablespace` para que reflita os novos locais (Se isto não for feito, o `pg_dump` continuará mostrando os locais antigos dos espaços de tabelas).

### Exemplo 18-2. Criação de espaço de tabelas no Windows

Este exemplo mostra o efeito produzido no diretório `$PGDATA/pg_tblspc` pela criação de um espaço de tabelas.<sup>10 11</sup>

```
=# CREATE TABLESPACE disco_f LOCATION 'F:\\postgresql';
```

```
CREATE TABLESPACE
```

```
=# SELECT * FROM pg_tablespace;
```

spcname	spcowner	spclocation	spcacl
pg_default	1		
pg_global	1		
disco_f	1	F:/postgresql	

(3 linhas)

```
E:\Program Files\PostgreSQL\8.0\data> dir pg_tblspc
```

```
Volume in drive E is Local Disk
Volume Serial Number is 1C2A-9875
```

```
Directory of E:\Program Files\PostgreSQL\8.0\data\pg_tblspc
```

```
21/06/2005  11:16      <DIR>          .
21/06/2005  11:16      <DIR>          ..
21/06/2005  11:16      <JUNCTION>      58588
                0 File(s)                0 bytes
                3 Dir(s)   3.744.190.464 bytes free
```

## Notas

1. Um *catálogo* é uma coleção nomeada de esquemas-SQL, descritores de servidores remotos e descritores de empacotadores de dados remotos em um ambiente-SQL. Os mecanismos para criar e remover os catálogos são definidos pela implementação. Um *esquema-SQL*, geralmente referido simplesmente como esquema, é uma coleção nomeada, persistente, de descritores. Qualquer objeto cujo descritor está em algum esquema-SQL é conhecido como um objeto do esquema-SQL. (ISO-ANSI Working Draft) Framework (SQL/Framework), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
2. Oracle 9i — O *esquema* é uma coleção nomeada de objetos, como tabelas, visões, agrupamentos, procedimentos e pacotes, associados a um determinado usuário. Oracle9i Database Administrator's Guide - Glossary (<http://www.stanford.edu/dept/itss/docs/oracle/9i/win.920/a95491/glossary.htm#432367>) (N. do T.)
3. DB2 8.1 — Os *esquemas* são objetos do banco de dados utilizados no DB2 para agrupar logicamente outros objetos de banco de dados. A maioria dos objetos de banco de dados têm seus nomes formados usando uma convenção para nomes de duas partes (nome\_do\_esquema.nome\_do\_objeto). A primeira parte do nome é referida como nome do esquema (também conhecida como qualificador do objeto de banco de dados). A segunda parte é o nome do objeto. DB2® Universal Database V8 for Linux, UNIX, and Windows Database Administration Certification Guide, 5th Edition (<http://www.phptr.com/title/0130463612>), George Baklarz e Bill Wong, Series IBM Press, Prentice Hall Professional Technical Reference, 2003, pág. 192. (N. do T.)
4. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
5. Oracle 9i — O banco de dados é dividido em uma ou mais unidades lógicas de armazenamento chamadas de *espaços de tabelas*. Os espaços de tabelas são divididos em unidades lógicas de armazenamento chamadas de segmentos, que por sua vez são divididas em extensões. Oracle9i Database Administrator's Guide - Glossary (<http://www.stanford.edu/dept/itss/docs/oracle/9i/win.920/a95491/glossary.htm#432418>) (N. do T.)
6. DB2 8.1 — Os *espaços de tabelas* são camadas lógicas entre o banco de dados e as tabelas armazenadas no banco de dados. Os espaços de tabelas são criados no banco de dados, e as tabelas são criadas no espaço de tabelas. DB2® Universal Database V8 for Linux, UNIX, and Windows Database Administration Certification Guide, 5th Edition (<http://www.phptr.com/title/0130463612>), George Baklarz e Bill Wong, Series IBM Press, Prentice Hall Professional Technical Reference, 2003, pág. 193. (N. do T.)

7. SQL Server 2000 — Um *grupo de arquivos* categoriza os arquivos do sistema operacional que contêm dados de um único banco de dados para simplificar as tarefas de administração do banco de dados, como a cópia de segurança. O grupo de arquivos é uma propriedade do banco de dados, não podendo conter arquivos do sistema operacional de mais de um banco de dados, embora um único banco de dados possa conter mais de um grupo de arquivos. Quando o banco de dados é criado, este é criado exatamente um grupo de arquivos chamado `PRIMARY`. Após a criação do banco de dados podem ser adicionados grupos de arquivos ao banco de dados. O nome do grupo de arquivos pode ser especificado nos comandos `CREATE TABLE` e `CREATE INDEX`, direcionando o armazenamento dos dados do banco de dados. SQL Server Books Online. (N. do T.)
8. unidade de estado sólido — um tipo de unidade de `hardware` que não contém partes móveis, geralmente esta unidade é feita em grande parte de circuitos eletrônicos. Solid-state device (<http://www.computerhope.com/jargon/s/solistat.htm>). (N. do T.)
9. O *vínculo simbólico* aponta para um arquivo por nome. Quando o `kernel` encontra um vínculo simbólico ao procurar por um nome de caminho, redireciona sua atenção para o nome de caminho armazenado como conteúdo do vínculo. Manual de Administração do Sistema Unix - Evi Nemeth e outros - Bookman. (N. do T.)
10. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
11. `junction point` — o ponto de junção é uma posição física no disco rígido que aponta para dados localizados em outro local no disco rígido ou em outra unidade de armazenamento. Os pontos de junção são criados quando é criada uma unidade montada. Os pontos de junção também podem ser criados pelo comando `linkd`. Microsoft Glossary for Business Users (<http://www.microsoft.com/atwork/glossary.msp>) (N. do T.)

# Capítulo 19. Autenticação de clientes

Quando um aplicativo cliente se conecta ao servidor de banco de dados especifica o nome de usuário do PostgreSQL a ser usado na conexão, de forma semelhante à feita pelo usuário para acessar o sistema operacional Unix. Dentro do ambiente SQL, o nome de usuário do banco de dados determina os privilégios de acesso aos objetos do banco de dados — Para obter mais informações deve ser visto o Capítulo 17. Portanto, é essencial controlar como os usuários de banco de dados podem se conectar.

A *autenticação* é o processo pelo qual o servidor de banco de dados estabelece a identidade do cliente e, por extensão, determina se o aplicativo cliente (ou o usuário executando o aplicativo cliente) tem permissão para se conectar com o nome de usuário que foi informado.

O PostgreSQL possui vários métodos diferentes para autenticação de clientes. O método utilizado para autenticar uma determinada conexão cliente pode ser selecionado tomando por base o endereço de hospedeiro (do cliente), o banco de dados ou o usuário.

Os nomes de usuário do PostgreSQL são logicamente distintos dos nomes de usuário do sistema operacional onde o servidor executa. Se todos os usuários de um determinado servidor de banco de dados também possuem conta no sistema operacional do servidor, é razoável atribuir nomes de usuário do banco de dados correspondendo aos nomes de usuário do sistema operacional. Entretanto, um servidor que aceita conexões remotas pode possuir muitos usuários de banco de dados que não possuem conta no sistema operacional local e, nestes casos, a associação entre os nomes de usuário do banco de dados e os nomes de usuário do sistema operacional não é necessária.

## 19.1. O arquivo `pg_hba.conf`

A autenticação do cliente é controlada pelo arquivo que por tradição se chama `pg_hba.conf` e é armazenado no diretório de dados do agrupamento de bancos de dados. HBA significa autenticação baseada no hospedeiro (*host-based authentication*). É instalado um arquivo `pg_hba.conf` padrão quando o diretório de dados é inicializado pelo utilitário `initdb`. Entretanto, é possível colocar o arquivo de configuração da autenticação em outro local; consulte o parâmetro de configuração `hba_file`.

O formato geral do arquivo `pg_hba.conf` é um conjunto de registros, sendo um por linha. As linhas em branco são ignoradas, da mesma forma que qualquer texto após o caractere de comentário `#`. Um registro é formado por vários campos separados por espaços ou tabulações. Os campos podem conter espaços em branco se o valor do campo estiver entre aspas. Os registros não podem ocupar mais de uma linha.

Cada registro especifica um tipo de conexão, uma faixa de endereços de IP de cliente (se for relevante para o tipo de conexão), um nome de banco de dados, um nome de usuário e o método de autenticação a ser utilizado nas conexões que correspondem a estes parâmetros. O primeiro registro com o tipo de conexão, endereço do cliente, banco de dados solicitado e nome de usuário que corresponder é utilizado para realizar a autenticação. Não existe *fall-through* (procura exaustiva) ou *backup*: se um registro for escolhido e a autenticação não for bem-sucedida, os próximos registros não serão levados em consideração. Se não houver correspondência com nenhum registro, então o acesso é negado.

O registro pode ter um dos sete formatos a seguir:

```
local      banco_de_dados usuário método_de_autenticação [opção_de_autenticação]
host       banco_de_dados usuário endereço_de_CIDR método_de_autenticação [opção_de_autenticação]
hostssl    banco_de_dados usuário endereço_de_CIDR método_de_autenticação [opção_de_autenticação]
hostnossl  banco_de_dados usuário endereço_de_CIDR método_de_autenticação [opção_de_autenticação]
host       banco_de_dados usuário endereço_de_IP máscara_de_IP método_de_autenticação
[opção_de_autenticação]
hostssl    banco_de_dados usuário endereço_de_IP máscara_de_IP método_de_autenticação
[opção_de_autenticação]
hostnossl  banco_de_dados usuário endereço_de_IP máscara_de_IP método_de_autenticação
[opção_de_autenticação]
```

O significado de cada campo está descrito abaixo:

**local**

Este registro corresponde às tentativas de conexão feitas utilizando soquete do domínio Unix. Sem um registro deste tipo não são permitidas conexões através de soquete do domínio Unix.

**host**

Este registro corresponde às tentativas de conexão feitas utilizando o protocolo TCP/IP. Os registros `host` correspondem tanto às conexões SSL (*Secure Socket Layer*), quanto às não SSL.

**Nota:** Não serão possíveis conexões TCP/IP remotas, a menos que o servidor seja inicializado com o valor apropriado para o parâmetro de configuração `listen_addresses`, uma vez que o comportamento padrão é aceitar conexões TCP/IP apenas no endereço retornante (`loopback`) `localhost`.

**hostssl**

Este registro corresponde às tentativas de conexão feitas utilizando o protocolo TCP/IP, mas somente quando a conexão é feita com a criptografia SSL.

Para esta opção poder ser utilizada o servidor deve ter sido construído com o suporte a SSL habilitado. Além disso, o SSL deve ser habilitado na inicialização do servidor através do parâmetro de configuração `ssl` (Para obter informações adicionais deve ser consultada a Seção 16.7).

**hostnssl**

Este registro é o oposto lógico de `hostssl`: só corresponde a tentativas de conexão feitas através do protocolo TCP/IP que não utilizam SSL.

**banco\_de\_dados**

Especifica quais bancos de dados este registro corresponde. O valor `all` especifica que corresponde a todos os bancos de dados. O valor `sameuser` especifica que o registro corresponde ao banco de dados com o mesmo nome do usuário fazendo o pedido de conexão. O valor `samegroup` especifica que o usuário deve ser membro do grupo com o mesmo nome do banco de dados do pedido de conexão. Senão, é o nome de um banco de dados específico do PostgreSQL. Podem ser fornecidos vários nomes de banco de dados separados por vírgula. Pode ser especificado um arquivo contendo nomes de banco de dados, precedendo o nome do arquivo por `@`.

**usuário**

Especifica quais usuários do PostgreSQL este registro corresponde. O valor `all` especifica que corresponde a todos os usuários. Senão, é o nome de um usuário específico do PostgreSQL. Podem ser fornecidos vários nomes de usuário separados por vírgula. Podem ser especificados nomes de grupo precedendo o nome do grupo por `+`. Pode ser especificado um arquivo contendo nomes de usuário precedendo o nome do arquivo por `@`.

**endereço\_de\_CIDR**

Especifica a faixa de endereços de IP da máquina cliente que este registro corresponde. Contém um endereço de IP na notação padrão decimal com pontos, e o comprimento da máscara de CIDR (Os endereços de IP somente podem ser especificados numericamente, e não como domínios ou nomes de hospedeiro). O comprimento da máscara indica o número de bits de mais alta ordem que o endereço de IP do cliente deve corresponder. Os bits à direita devem ser zero em um determinado endereço de IP. Não pode haver espaços em branco entre os endereços de IP, a `/` e o comprimento da máscara de CIDR.<sup>1</sup>

Um `endereço_de_CIDR` típico seria `172.20.143.89/32` para um único hospedeiro, ou `172.20.143.0/24` para uma rede. Para especificar um único hospedeiro deve ser utilizada uma máscara de CIDR igual a 32 para o IPv4, ou igual a 128 para o IPv6.

Um endereço especificado no formato IPv4 corresponde às conexões IPv6 que possuem o endereço correspondente como, por exemplo, `127.0.0.1` corresponde ao endereço de IPv6 `::ffff:127.0.0.1`. Uma entrada especificada no formato IPv6 corresponde apenas às conexões IPv6, mesmo que represente um endereço na faixa IPv4-em-IPv6. Deve ser observado que as entradas no formato IPv6 serão rejeitadas se a biblioteca C do sistema não possuir suporte a endereços IPv6.

Este campo se aplica apenas aos registros `host`, `hostssl` e `hostnssl`.

*endereço\_de\_IP**máscara\_de\_IP*

Estes campos podem ser utilizados como uma alternativa à notação *endereço\_de\_CIDR*. Em vez de especificar o comprimento da máscara, a máscara é especificada como uma coluna em separado. Por exemplo, 255.0.0.0 representa uma máscara de CIDR para endereços de IPv4 com comprimento igual a 8, e 255.255.255.255 representa uma máscara de CIDR com comprimento igual a 32.

Estes campos se aplicam apenas aos registros *host*, *hostssl* e *hostnossl*.

*método\_de\_autenticação*

Especifica o método de autenticação a ser utilizado para se conectar através deste registro. Abaixo está mostrado um resumo das escolhas possíveis; os detalhes podem ser encontrados na Seção 19.2.

*trust*

A conexão é permitida incondicionalmente. Este método permite a qualquer um que possa se conectar ao servidor de banco de dados PostgreSQL se autenticar como o usuário do PostgreSQL que for desejado, sem necessidade de senha. Consulte a Seção 19.2.1 para obter detalhes.

*reject*

A conexão é rejeitada incondicionalmente. É útil para “eliminar por filtragem” certos hospedeiros de um grupo.

*md5*

Requer que o cliente forneça uma senha criptografada pelo método *md5* para autenticação. Consulte a Seção 19.2.2 para obter detalhes.

*crypt*

Requer que o cliente forneça uma senha criptografada através de *crypt()* para autenticação. Deve-se dar preferência ao método *md5* para os clientes com versão 7.2 ou posterior, mas os clientes com versão anterior a 7.2 somente suportam *crypt*. Consulte a Seção 19.2.2 para obter detalhes.

*password*

Requer que o cliente forneça uma senha não criptografada para autenticação. Uma vez que a senha é enviada em texto puro pela rede, não deve ser utilizado em redes não confiáveis. Consulte a Seção 19.2.2 para obter detalhes.

*krb4*

É utilizado Kerberos V4 para autenticar o usuário. Somente disponível para conexões TCP/IP. Consulte a Seção 19.2.3 para obter detalhes.

*krb5*

É utilizado Kerberos V5 para autenticar o usuário. Somente disponível para conexões TCP/IP. Consulte a Seção 19.2.3 para obter detalhes.

*ident*

Obtém o nome de usuário do sistema operacional do cliente (para conexões TCP/IP fazendo contato com o servidor de identificação no cliente, para conexões locais obtendo a partir do sistema operacional) e verifica se o usuário possui permissão para se conectar como o usuário de banco de dados solicitado consultando o mapa especificado após a palavra chave *ident*. Consulte a Seção 19.2.4 para obter detalhes.

*pam*

Autenticação utilizando o serviço Pluggable Authentication Modules (PAM) fornecido pelo sistema operacional. Consulte a Seção 19.2.5 para obter detalhes.

*opção\_de\_autenticação*

O significado deste campo opcional depende do método de autenticação escolhido, estando descrito na próxima seção.

Os arquivos incluídos pela construção @ são lidos como nomes de listas, que podem ser separadas por espaços em branco ou vírgulas. Os comentários são iniciados por #, como no arquivo *pg\_hba.conf*, sendo permitidas construções @



aninhadas. A menos que o nome do arquivo que segue a @ seja um caminho absoluto, é considerado como sendo relativo ao diretório que contém o arquivo que faz referência.

Uma vez que os registros de `pg_hba.conf` são examinados sequencialmente a cada tentativa de conexão, a ordem dos registros possui significado. Normalmente, os primeiros registros possuem parâmetros de correspondência de conexão mais exigentes e métodos de autenticação menos exigentes, enquanto os últimos registros possuem parâmetros de correspondência menos exigentes e métodos de autenticação mais exigentes. Por exemplo, pode-se desejar utilizar a autenticação `trust` para conexões TCP/IP locais, mas requerer o uso de senha para conexões TCP/IP remotas. Neste caso, o registro especificando a autenticação `trust` para conexões a partir de 127.0.0.1 deve aparecer antes do registro especificando autenticação por senha para uma faixa mais ampla de endereços de IP de cliente permitidos.

O arquivo `pg_hba.conf` é lido durante a inicialização e quando o processo servidor principal (`postmaster`) recebe um sinal `SIGHUP`. Se o arquivo for editado enquanto o sistema estiver ativo, será necessário enviar um sinal para o `postmaster` (utilizando `pg_ctl reload` ou `kill -HUP`) para fazer com que o arquivo seja lido novamente.

No Exemplo 19-1 são mostrados alguns exemplos de registros do arquivo `pg_hba.conf`. Para obter detalhes sobre os diferentes métodos de autenticação deve ser consultada a próxima seção.

### Exemplo 19-1. Exemplo de registros do arquivo `pg_hba.conf`

```
# Permitir qualquer usuário do sistema local se conectar a qualquer banco
# de dados sob qualquer nome de usuário utilizando os soquetes do domínio
# Unix (o padrão para conexões locais).
#
# TYPE      DATABASE      USER      CIDR-ADDRESS      METHOD
local      all             all                          trust

# A mesma coisa utilizando conexões locais TCP/IP retornantes (loopback).
#
# TYPE      DATABASE      USER      CIDR-ADDRESS      METHOD
host       all             all        127.0.0.1/32      trust

# O mesmo que o exemplo anterior mas utilizando uma coluna em separado para
# máscara de rede.
#
# TYPE      DATABASE      USER      IP-ADDRESS      IP-MASK      METHOD
host       all             all        127.0.0.1      255.255.255.255      trust

# Permitir qualquer usuário de qualquer hospedeiro com endereço de IP 192.168.93.x
# se conectar ao banco de dados "template1" com o mesmo nome de usuário que "ident"
# informa para a conexão (normalmente o nome de usuário do Unix).
#
# TYPE      DATABASE      USER      CIDR-ADDRESS      METHOD
host       template1    all        192.168.93.0/24    ident sameuser

# Permitir o usuário do hospedeiro 192.168.12.10 se conectar ao banco de dados
# "template1" se a senha do usuário for fornecida corretamente.
#
# TYPE      DATABASE      USER      CIDR-ADDRESS      METHOD
host       template1    all        192.168.12.10/32    md5

# Na ausência das linhas "host" precedentes, estas duas linhas rejeitam todas
# as conexões oriundas de 192.168.54.1 (uma vez que esta entrada será
# correspondida primeiro), mas permite conexões Kerberos V de qualquer ponto
# da Internet. A máscara zero significa que não é considerado nenhum bit do
# endereço de IP do hospedeiro e, portanto, corresponde a qualquer hospedeiro.
#
# TYPE      DATABASE      USER      CIDR-ADDRESS      METHOD
host       all             all        192.168.54.1/32    reject
host       all             all        0.0.0.0/0          krb5

# Permite os usuários dos hospedeiros 192.168.x.x se conectarem a qualquer
# banco de dados se passarem na verificação de "ident". Se, por exemplo, "ident"
```

```
# informar que o usuário é "oliveira" e este requerer se conectar como o usuário
# do PostgreSQL "guest1", a conexão será permitida se houver uma entrada
# em pg_ident.conf para o mapa "omicron" informando que "oliveira" pode se
# conectar como "guest1".
#
# TYPE    DATABASE    USER        CIDR-ADDRESS    METHOD
host      all           all          192.168.0.0/16   ident omicron

# Se as linhas abaixo forem as únicas três linhas para conexão local, vão
# permitir os usuários locais se conectarem somente aos seus próprios bancos de
# dados (bancos de dados com o mesmo nome que seus nomes de usuário), exceto
# para os administradores e membros do grupo "suporte" que podem se conectar a
# todos os bancos de dados. O arquivo $PGDATA/admins contém a lista de nomes de
# usuários. A senha é requerida em todos os casos.
#
# TYPE    DATABASE    USER        CIDR-ADDRESS    METHOD
local     sameuser      all          md5
local     all            @admins      md5
local     all            +suporte     md5

# As duas últimas linhas acima podem ser combinadas em uma única linha:
local     all            @admins,+suporte     md5

# A coluna banco de dados também pode utilizar listas e nomes de arquivos,
# mas não grupos:
local     db1,db2,@demodbs all          md5
```

## 19.2. Métodos de autenticação

Abaixo estão descritos com mais detalhes os métodos de autenticação.

### 19.2.1. Autenticação de confiança

Quando é especificada a autenticação `trust`, o PostgreSQL assume que qualquer um que possa se conectar ao servidor está autorizado a acessar o banco de dados como qualquer usuário que seja especificado (incluindo o superusuário do banco de dados). É claro que as restrições especificadas para o usuário e para o banco de dados ainda se aplicam. Este método deve ser utilizado somente quando existe uma proteção adequada no nível de sistema operacional para as conexões ao servidor.

A autenticação `trust` é apropriada e muito conveniente para conexões locais em estações de trabalho com um único usuário. Geralmente *não* é apropriada em uma máquina multiusuária. Entretanto, é possível utilizar `trust` mesmo em uma máquina multiusuária, se for restringido o acesso ao arquivo de soquete do domínio Unix do servidor utilizando permissões do sistema de arquivos. Para fazer isto, deve ser definido o parâmetro de configuração `unix_socket_permissions` (e, possivelmente, `unix_socket_group`) conforme descrito na Seção 16.4.2. Também pode ser definido o parâmetro de configuração `unix_socket_directory` para colocar o arquivo de soquete em um diretório com restrição adequada.

Definir as permissões do sistema de arquivos somente serve para as conexões através dos soquetes do Unix. As conexões locais TCP/IP não são restringidas desta maneira; portanto, se for desejado utilizar permissões do sistema de arquivos para segurança local, deve ser removida a linha `host ... 127.0.0.1 ...` do arquivo `pg_hba.conf`, ou mudá-la para que use um método de autenticação diferente de `trust`.

A autenticação `trust` somente é adequada para as conexões TCP/IP quando se confia em todos os usuários de todas as máquinas que podem se conectar ao servidor de banco de dados pelas linhas do arquivo `pg_hba.conf` que especificam `trust`. Raramente faz sentido utilizar `trust` para conexões TCP/IP, a não ser as oriundas de `localhost` (127.0.0.1).

### 19.2.2. Autenticação por senha

Os métodos de autenticação baseados em senha são `md5`, `crypt` e `password`. Estes métodos operam de forma semelhante, exceto com relação à forma como a senha é enviada através da conexão, mas somente o método `md5` suporta senhas criptografadas armazenadas no catálogo do sistema `pg_shadow`; os outros dois métodos requerem que sejam armazenadas senhas não criptografadas neste catálogo.

Se houver preocupação com relação aos ataques de “farejamento” (sniffing) de senhas, então md5 é o método preferido, com crypt como a segunda opção se for necessário suportar clientes pré-7.2. O método password deve ser evitado, especialmente em conexões pela Internet aberta (a menos que seja utilizado SSL, SSH ou outro método de segurança para proteger a conexão).

As senhas de banco de dados do PostgreSQL são distintas das senhas de usuário do sistema operacional. As senhas de todos os usuários do banco de dados são armazenadas na tabela do catálogo do sistema pg\_shadow. As senhas podem ser gerenciadas através dos comandos SQL *CREATE USER* e *ALTER USER*; por exemplo, **CREATE USER foo WITH PASSWORD 'segredo' ;** Por padrão, ou seja, se nenhuma senha for definida, é armazenado o valor nulo para a senha e a autenticação da senha é sempre mal-sucedida para este usuário.

### 19.2.3. Autenticação Kerberos

Kerberos é um sistema de autenticação seguro, padrão da indústria, adequado para computação distribuída em redes públicas. A descrição do sistema Kerberos está muito acima do escopo deste documento; de forma geral pode ser bastante complexo (porém poderoso). As páginas Kerberos: The Network Authentication Protocol (<http://web.mit.edu/kerberos/>) e MIT Project Athena (<ftp://athena-dist.mit.edu>) podem ser um bom ponto de partida para estudá-lo. Existem diversas fontes de distribuição do Kerberos.

Embora o PostgreSQL suporte tanto o Kerberos 4 quanto o Kerberos 5, somente o Kerberos 5 é recomendado. O Kerberos 4 é considerado inseguro, não sendo mais recomendado para uso geral.

Para o Kerberos poder ser utilizado, o seu suporte deve ser habilitado em tempo de construção. Para obter mais informações deve ser visto o Capítulo 14. São suportados tanto o Kerberos 4 quanto 5, mas pode ser suportada apenas uma versão por uma construção.

O PostgreSQL opera como um serviço Kerberos normal. O nome do serviço principal é *nome\_do\_serviço/nome\_do\_hospedeiro@domínio*, onde *nome\_do\_serviço* é postgres (a menos que seja selecionado um nome de serviço diferente em tempo de configuração utilizando `./configure --with-krb-srvnam=qualquer_coisa`). O *nome\_do\_hospedeiro* é o nome de hospedeiro da máquina servidora totalmente qualificado. O domínio (*realm*) do serviço principal é o domínio preferido da máquina servidora.

Os principais <sup>2</sup> dos clientes devem ter seu nome de usuário do PostgreSQL como primeiro componente como, por exemplo, *pgusername/outras\_coisas@dominio*. Atualmente o domínio do cliente não é verificado pelo PostgreSQL; portanto, estando habilitada a autenticação entre domínios, então todo principal de qualquer domínio que puder se comunicar com o que está sendo utilizado será aceito.

Certifique-se que o arquivo de chaves do servidor é legível (e preferencialmente somente legível) pela conta do servidor PostgreSQL (Consulte também a Seção 16.1). O local do arquivo de chaves é especificado pelo parâmetro de configuração em tempo de execução *krb\_server\_keyfile* (Consulte também a Seção 16.4). O padrão é */etc/srvtab* se estiver sendo utilizado o Kerberos 4, e */usr/local/pgsql/etc/krb5.keytab* (o ou diretório especificado como *sysconfdir* em tempo de construção) no Kerberos 5.

Para gerar o arquivo keytab deve ser utilizado, por exemplo (com a versão 5):

```
kadmin% ank -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

Para obter detalhes deve ser lida a documentação do Kerberos.

Ao se conectar ao banco de dados tenha certeza de possuir um tíquete para o principal correspondendo ao nome de usuário do banco de dados. Como exemplo: Para o nome de usuário do banco de dados *cecilia*, podem ser utilizados tanto o principal *cecilia@EXAMPLE.COM* quanto *cecilia/users.example.com@EXAMPLE.COM* para se autenticar no servidor de banco de dados.

Se for utilizado no servidor Web Apache o módulo *mod\_auth\_kerb* de Kerberos Module for Apache (<http://modauthkerb.sf.net>) e o *mod\_perl*, pode ser utilizado *AuthType KerberosV5SaveCredentials* com um script *mod\_perl*, proporcionando um acesso seguro ao banco de dados através da Web, sem necessidade de senhas adicionais.

### 19.2.4. Autenticação baseada no Ident

O método de autenticação *ident* funciona obtendo o nome de usuário do sistema operacional do cliente, e determinando os nomes de usuário do banco de dados permitidos utilizando um arquivo de mapa que lista os pares de nomes de usuários

correspondentes permitidos. A determinação do nome de usuário do cliente é o ponto crítico da segurança, funcionando de forma diferente dependendo do tipo de conexão.

#### 19.2.4.1. Autenticação Ident através de TCP/IP

O “Protocolo de Identificação” está descrito no *RFC 1413*. Virtualmente todo sistema operacional da família Unix é distribuído com um servidor `ident` que atende a porta TCP 113 por padrão. A funcionalidade básica do servidor `ident` é responder a perguntas como “Que usuário inicializou a conexão que sai pela sua porta *X* e se conecta à minha porta *Y*?”. Uma vez que o PostgreSQL conhece tanto *X* quanto *Y* quando a conexão física é estabelecida, pode fazer a pergunta ao servidor `ident` no hospedeiro do cliente se conectando e pode, teoricamente, determinar o usuário do sistema operacional para uma determinada conexão desta maneira.

O problema deste procedimento é que depende da integridade do cliente: se a máquina cliente não for confiável ou estiver comprometida, alguém querendo fazer um ataque pode executar algum programa na porta 113 e retornar qualquer nome de usuário desejado. Este método de autenticação é, portanto, apropriado apenas para redes fechadas onde toda máquina cliente está sob controle rígido, e onde os administradores do sistema e do banco de dados trabalham de forma integrada. Em outras palavras, é necessário confiar na máquina executando o servidor `ident`. Deve ser observada a advertência:

RFC 1413

O Protocolo de Identificação não tem por objetivo ser um protocolo de autorização ou de controle de acesso.

#### 19.2.4.2. Autenticação Ident através de soquetes locais

Nos sistemas que possuem a opção `SO_PEERCREDS`<sup>3</sup> para os soquetes do domínio Unix (atualmente Linux, FreeBSD, NetBSD, OpenBSD e BSD/OS), a autenticação `ident` também pode ser aplicada para as conexões locais. Neste caso nenhum risco à segurança é introduzido pela utilização da autenticação `ident`; na verdade, esta é a escolha preferida para as conexões locais nestes sistemas.

Em sistemas que não possuem a opção `SO_PEERCREDS`, a autenticação `ident` está disponível somente através das conexões TCP/IP. Para superar este problema, é possível especificar o endereço 127.0.0.1 do localhost, e se conectar a este endereço. Este método é tão confiável quanto o servidor `ident` local.

#### 19.2.4.3. Mapas de Ident

Ao utilizar a autenticação baseada no `ident`, após determinar o nome de usuário do sistema operacional que iniciou a conexão o PostgreSQL verifica se este usuário pode se conectar como o usuário de banco de dados que está sendo solicitado na conexão. Isto é controlado pelo argumento do mapa de `ident` que segue a palavra chave `ident` no arquivo `pg_hba.conf`. Existe um mapa de `ident` pré-definido chamado `sameuser`, que permite todo usuário do sistema operacional se conectar como o usuário de banco de dados com o mesmo nome (se este existir). Os outros mapas devem ser criados manualmente.

Fora o `sameuser`, os mapas de `ident` são definidos no arquivo de mapa de `ident` que, por padrão, se chama `pg_ident.conf` e é armazenado no diretório de dados do agrupamento (Entretanto, é possível colocar o arquivo de mapas outro lugar; consulte o parâmetro de configuração `ident_file`). A forma geral das linhas do mapa de `ident` é:

```
nome_do_mapa nome_de_usuario_do_ident nome_de_usuario_do_banco_de_dados
```

Os comentários e os espaços em branco são tratados da mesma maneira que no arquivo `pg_hba.conf`. O `nome_do_mapa` é um nome arbitrário a ser utilizado para fazer referência ao mapa em `pg_hba.conf`. Os outros dois campos especificam qual usuário do sistema operacional pode se conectar como qual usuário do banco de dados. O mesmo `nome_do_mapa` pode ser utilizado várias vezes para especificar mais mapeamentos de usuários dentro de um único mapa. Não existe restrição com relação a quantos usuários do banco de dados um determinado usuário do sistema operacional pode corresponder, nem o contrário.

O arquivo `pg_ident.conf` é lido na inicialização e quando o processo servidor principal (`postmaster`) recebe um sinal `SIGHUP`. Se o arquivo for editado com o sistema ativo, será necessário enviar este sinal para o `postmaster` (utilizando `pg_ctl reload` ou `kill -HUP`) para fazer o arquivo ser lido novamente.

Um arquivo `pg_ident.conf` que pode ser utilizado em conjunto com o arquivo `pg_hba.conf` do Exemplo 19-1 está mostrado no Exemplo 19-2. Nesta configuração de exemplo, qualquer usuário autenticado em uma máquina da rede 192.168 que não possua o nome de usuário Unix `oliveira`, `lia` ou `andre` não vai ter o acesso permitido. O usuário Unix `andre` somente poderá acessar quando tentar se conectar como o usuário do PostgreSQL `pacheco`, e não como `andre` ou

algum outro. A usuária lia somente poderá se conectar como lia. O usuário oliveira poderá se conectar como o próprio oliveira ou como guest1.

### Exemplo 19-2. Arquivo pg\_ident.conf de exemplo

```
# MAPNAME      IDENT-USERNAME  PG-USERNAME

omicron        oliveira        oliveira
omicron        lia             lia
# pacheco possui o nome de usuário andre nestas máquinas
omicron        andre           pacheco
# oliveira também pode se conectar como guest1
omicron        oliveira        guest1
```

## 19.2.5. Autenticação PAM

Este método de autenticação opera de forma semelhante ao método password, exceto por utilizar o mecanismo de autenticação PAM (Pluggable Authentication Modules). O nome padrão do serviço PAM é postgresql. É possível, opcionalmente, fornecer um outro nome de serviço após a palavra chave pam no arquivo pg\_hba.conf. Para obter informações adicionais sobre o PAM deve ser consultada a Página Linux-PAM (<http://www.kernel.org/pub/linux/libs/pam/>) e a Documentação do PAM do Solaris (<http://www.sun.com/software/solaris/pam/>).

## 19.3. Problemas de autenticação

Falhas de autenticação genuínas e problemas correlatos geralmente se manifestam através de mensagens de erro como as que se seguem.

```
FATAL:  no pg_hba.conf entry for host "123.123.123.123", user "andre", database "teste"
```

Tradução:

```
FATAL:  nenhuma entrada em pg_hba.conf para o hospedeiro "123.123.123.123", usuário
"andre", banco de dados "teste"
```

Esta é a mensagem mais provável de ocorrer quando o contato com o servidor for bem-sucedido, mas este não deseja falar com o cliente. Como a própria mensagem sugere, o servidor recusou o pedido de conexão porque não encontrou uma entrada autorizando esta conexão no arquivo de configuração pg\_hba.conf.

```
FATAL:  Password authentication failed for user "andre"
```

Tradução:

```
FATAL:  A autenticação da senha para o usuário "andre" não foi bem-sucedida
```

As mensagens deste tipo indicam que o servidor foi contactado, e este deseja se comunicar com o cliente, mas não até que se passe pelo método de autorização especificado no arquivo pg\_hba.conf. Deve ser verificada a senha fornecida, ou verificado o programa Kerberos ou ident se for mencionado algum destes tipos de autenticação na mensagem de erro.

```
FATAL:  user "andre" does not exist
```

Tradução:

```
FATAL:  o usuário "andr " n o existe
```

O nome de usu rio indicado n o foi encontrado.

```
FATAL:  database "teste" does not exist
```

Trad   o:

```
FATAL:  o banco de dados "teste" n o existe
```

Tentativa de conectar a um banco de dados que não existe. Deve ser observado que se não for especificado o nome do banco de dados, por padrão é usado o banco de dados com o mesmo nome do usuário, o que poder ser correto ou não.

**Dica:** O `log` do servidor pode conter informações adicionais sobre uma falha de autenticação do que o informado ao cliente. Havendo incerteza sobre o motivo da falha, o `log` deve ser verificado.

## Notas

1. `Classless Internet Domain Routing` — O CIDR, definido pela RFC1519, elimina o sistema de classes que determinava originalmente a parte de rede de um endereço IP. Como a sub-rede, da qual é uma extensão direta, o CIDR conta com uma máscara de rede explícita para definir o limite entre as partes de rede e de hospedeiro de um endereço. Exemplo: Comprimento: /20; Bits do hospedeiro: 12; Hospedeiros por rede: 4096; Máscara de rede decimal: 255.255.240.0. Manual de Administração do Sistema Unix - Evi Nemeth e outros - Bookman. (N. do T.)
2. O Kerberos é um sistema de autenticação de terceiros confiável, cuja finalidade principal é permitir pessoas e processos (conhecidos no Kerberos como principais) provarem suas identidades de uma maneira confiável através de redes não seguras. Em vez de transmitir as senhas secretas em aberto, que podem ser interceptadas e lidas por pessoas não autorizadas, os principais obtêm `vouchers` (conhecidos como tíquetes) do Kerberos, utilizados para se autenticar. Kerberos Concepts and Terms (<http://www.net.berkeley.edu/kerberos/k5concepts.html>). (N. do T.)
3. `SO_PEERCRECRED` — consulte `man 7 socket`. (N. do T.)

# Capítulo 20. Idioma

Este capítulo descreve as funcionalidades de idioma (`locale`) disponíveis do ponto de vista do administrador. O suporte a idioma no PostgreSQL é realizado de duas maneiras:

- Utilizando as funcionalidades de idioma do sistema operacional, para fornecer ordem de classificação, formatação de números, tradução das mensagens, e outros aspectos específicos do idioma.
- Disponibilizando no servidor PostgreSQL vários conjuntos de caracteres diferentes, incluindo conjuntos de caracteres de vários bytes, para permitir o armazenamento de textos em todos os idiomas, e provendo a tradução de conjuntos de caracteres entre o cliente e o servidor.

## 20.1. Suporte a idioma

O suporte a *idioma* se refere a um aplicativo que respeita as preferências culturais com relação ao alfabeto, classificação, formatação de números, etc. O PostgreSQL utiliza as facilidades de idioma ISO C e POSIX padrão fornecidas pelo sistema operacional do servidor. Para obter informações adicionais deve ser consultada a documentação do sistema utilizado.

### 20.1.1. Visão geral

O suporte a idioma é inicializado, automaticamente, quando o agrupamento de bancos de dados é criado utilizando o utilitário `initdb`. Por padrão, o `initdb` inicializa o agrupamento de bancos de dados com a definição de idioma do ambiente onde executa; portanto, se o sistema operacional estiver definido para utilizar o mesmo idioma desejado para o agrupamento de bancos de dados, então não é necessário ser feito mais nada. Se for desejado utilizar um idioma diferente (ou não houver certeza do idioma definido no sistema operacional), pode ser informado ao `initdb` qual é o idioma desejado através da opção `--locale`. Por exemplo:

```
initdb --locale=pt_BR
```

Este exemplo define o idioma como Português (`pt`) conforme falado no Brasil (`BR`). Outras possibilidades são `en_US` (Inglês dos Estados Unidos) e `fr_CA` (Francês do Canadá). Se o idioma permitir utilizar mais de um conjunto de caracteres, então a especificação ficará parecida com esta: `pt_BR.ISO8859-1`. Quais idiomas estão disponíveis no sistema operacional, e quais são os seus nomes, depende do que é disponibilizado pelo distribuidor do sistema operacional, e do que foi instalado (Na maioria dos sistemas o comando `locale -a` mostra a relação de idiomas disponíveis).

Ocasionalmente é útil combinar regras de idiomas diferentes como, por exemplo, regras de classificação do Inglês com mensagens em Português. Para que isto seja possível, existe um conjunto de subcategorias de idioma controlando somente certos aspectos das regras de idioma.

<code>LC_COLLATE</code>	Ordem de classificação das cadeias de caracteres <sup>a b</sup>
<code>LC_CTYPE</code>	Classificação dos caracteres (O que é uma letra? Sua letra maiúscula equivalente?) <sup>c</sup>
<code>LC_MESSAGES</code>	Idioma das mensagens
<code>LC_MONETARY</code>	Formatação das quantias monetárias
<code>LC_NUMERIC</code>	Formatação dos números
<code>LC_TIME</code>	Formatação das datas e das horas
<p>Notas:</p> <p>a. <code>collation; collating sequence</code> — Um método para comparar duas cadeias de caracteres comparáveis. Todo conjunto de caracteres possui seu <code>collation</code> padrão. (Second Informal Review Draft) ISO/IEC 9075:1992, Database Language SQL- July 30, 1992. (N. do T.)</p> <p>b. <code>SQL Server</code> — <code>collation</code>: se refere ao conjunto de regras que determinam como os dados são classificados e comparados. Microsoft SQL Server 2000 Introduction - Part No. X05-88268. (N. do T.)</p> <p>c. <code>LC_CTYPE</code> — Define a classificação do caractere, conversão maiúscula/minúscula, e outros atributos do caractere. <code>LC_CTYPE Category for the Locale Definition Source File Format</code> (<a href="http://publibn.boulder.ibm.com/doc_link/en_US/a_doc_lib/files/aixfiles/LC_CTYPE.htm">http://publibn.boulder.ibm.com/doc_link/en_US/a_doc_lib/files/aixfiles/LC_CTYPE.htm</a>) (N. do T.)</p>	

No utilitário `initdb` os nomes das categorias se traduzem em nomes de opção que mudam a escolha de idioma para uma determinada categoria. Por exemplo, para definir o idioma como sendo Francês do Canadá, mas utilizar as regras dos E.U.A para formatar valores monetários, deve ser utilizado `initdb --locale=fr_CA --lc-monetary=en_US`.

Se for desejado que o sistema se comporte como não tendo suporte a idioma, devem ser utilizados os idiomas especiais `C` ou `POSIX`.

Os valores de algumas categorias de idioma devem permanecer fixos por toda a existência do agrupamento de bancos de dados. Ou seja, uma vez executado o utilitário `initdb` não pode mais haver alteração dos valores definidos para estas categorias. Estas categorias são `LC_COLLATE` e `LC_CTYPE`, que afetam a ordem de classificação dos índices e, portanto, devem permanecer fixas ou os índices das colunas de texto vão ficar corrompidos. O PostgreSQL impõe esta restrição registrando os valores de `LC_COLLATE` e `LC_CTYPE` usados pelo `initdb`. O servidor adota estes dois valores, automaticamente, na inicialização.

Quando o servidor está em execução as demais categorias de idioma podem ser alteradas como se desejar, definindo as variáveis de configuração em tempo de execução que possuem o mesmo nome das categorias de idioma (consulte a Seção 16.4.8.2 para obter detalhes). Na verdade, os padrões escolhidos pelo utilitário `initdb` são escritos no arquivo de configuração `postgresql.conf` apenas para servirem como padrão quando o servidor é inicializado. Se forem removidas as atribuições presentes no arquivo `postgresql.conf`, o servidor herdará as definições do ambiente de execução.

Deve ser observado que o comportamento de idioma do servidor é determinado pelas variáveis de ambiente enxergadas pelo servidor, e não pelo ambiente de qualquer um dos clientes. Portanto, antes de inicializar o servidor deve-se tomar o cuidado de configurar as definições de idioma corretamente. Uma consequência deste fato é que, se o cliente e o servidor forem configurados com idiomas diferentes, as mensagens poderão ser mostradas em idiomas diferentes dependendo de onde forem originadas.

**Nota:** Quando se fala em herdar o idioma do ambiente de execução, isto significa o seguinte na maioria dos sistemas operacionais: Para uma determinada categoria de idioma, como o agrupamento, as seguintes variáveis de ambiente são consultadas, nesta ordem, até ser encontrada uma com valor definido: `LC_ALL`, `LC_COLLATE` (a variável correspondente à respectiva categoria) e `LANG`. Se nenhuma destas variáveis de ambiente estiver definida, então o padrão é utilizar o idioma `C`.

Algumas bibliotecas de idioma de mensagem também examinam a variável de ambiente `LANGUAGE`, que prevalece sobre todas as outras definições de idioma para a finalidade de definir o idioma das mensagens. Em caso de dúvida, deve ser consultada a documentação do sistema operacional para obter informações adicionais, em particular a documentação sobre `gettext`.

Para habilitar as mensagens traduzidas para o idioma preferido do usuário, deve ser habilitado o NLS (suporte a idioma nacional) em tempo de construção. Esta escolha independe dos outros suportes a idioma.

## 20.1.2. Comportamento

O suporte a idioma exerce influência sobre as seguintes funcionalidades:

- Ordem de classificação das consultas que utilizam a cláusula `ORDER BY`.
- A capacidade de utilizar índices com a cláusula `LIKE`.
- A família de funções `to_char`.

A desvantagem de utilizar idiomas diferentes de `C` e `POSIX` no PostgreSQL é o impacto no desempenho. Torna a manipulação de caracteres mais lenta, e impede a utilização de índices comuns na cláusula `LIKE`. *Por estes motivos, a utilização de idioma deve ser feita somente quando for realmente necessária.*

## 20.1.3. Problemas

Se, apesar do que foi explicado acima, o suporte a idioma não funcionar, deve ser verificado no sistema operacional se o suporte a idioma está configurado de forma correta. Pode ser utilizado o comando `locale -a` para verificar quais idiomas estão instalados no sistema operacional, caso este comando esteja disponível no sistema operacional utilizado.<sup>1</sup>

Deve ser verificado se o PostgreSQL está realmente utilizando o idioma que se pensa que esteja utilizando. As definições de `LC_COLLATE` e `LC_CTYPE` são determinadas quando o utilitário `initdb` é executado, não podendo ser mudadas sem que o `initdb` seja executado novamente. Outras definições de idioma, incluindo `LC_MESSAGES` e `LC_MONETARY`, são



determinadas inicialmente a partir do ambiente onde o servidor é posto em execução. As definições ativas de idioma podem ser verificadas utilizando o comando `SHOW`.

Na distribuição do código fonte, o diretório `src/test/locale` contém um conjunto de testes para o suporte a idioma do PostgreSQL.

Como é óbvio, os aplicativos cliente que tratam os erros gerados pelo servidor analisando o texto da mensagem de erro terão problemas quando as mensagens do servidor estiverem em outro idioma. Os autores destes aplicativos são aconselhados a utilizar o esquema de códigos de erro para tratar erros, em vez dos textos das mensagens.

A manutenção dos catálogos de mensagens traduzidas requer o esforço contínuo de muitos voluntários que desejam ver o PostgreSQL falando bem o seu idioma. Se as mensagens no seu idioma não estiverem disponíveis atualmente, ou se não estiverem inteiramente traduzidas, sua ajuda será apreciada. Se desejar ajudar, consulte o Capítulo 44 ou escreva para a lista de discussão dos desenvolvedores.

## 20.2. Suporte a conjuntos de caracteres

No PostgreSQL o suporte a conjuntos de caracteres permite armazenar textos usando vários conjuntos de caracteres, incluindo conjuntos de caracteres de um único byte, como a série ISO 8859, e conjuntos de caracteres de vários bytes, como o EUC (Extended Unix Code), Unicode e o código interno Mule. Todos os conjuntos de caracteres podem ser utilizados de forma transparente no servidor (Se forem utilizadas funções de extensão de terceiros, vai depender do código destas funções utilizadas ter sido escrito corretamente). O conjunto de caracteres padrão é selecionado durante a inicialização do agrupamento de bancos de dados do PostgreSQL, feito pelo utilitário `initdb`. Ao se criar um banco de dados através do utilitário `createdb`, ou através do comando `CREATE DATABASE` da linguagem SQL, pode ser escolhido um conjunto de caracteres diferente do padrão do agrupamento. Portanto, podem haver vários bancos de dados, cada um com um conjunto de caracteres diferentes.

### 20.2.1. Conjuntos de caracteres aceitos

A Tabela 20-1 mostra os conjuntos de caracteres disponíveis para uso no servidor.

**Tabela 20-1. Conjuntos de caracteres do servidor**

Nome	Descrição
SQL_ASCII	ASCII
EUC_JP	EUC Japonês
EUC_CN	EUC Chinês
EUC_KR	EUC Coreano
JOHAB	EUC Coreano (baseado em Hangle)
EUC_TW	EUC Tailandês
UNICODE	Unicode (UTF-8) <sup>a</sup>
MULE_INTERNAL	Código interno Mule
LATIN1	ISO 8859-1/ECMA 94 (Alfabeto latino nº 1)
LATIN2	ISO 8859-2/ECMA 94 (Alfabeto latino nº 2)
LATIN3	ISO 8859-3/ECMA 94 (Alfabeto latino nº 3)
LATIN4	ISO 8859-4/ECMA 94 (Alfabeto latino nº 4)
LATIN5	ISO 8859-9/ECMA 128 (Alfabeto latino nº 5)
LATIN6	ISO 8859-10/ECMA 144 (Alfabeto latino nº 6)
LATIN7	ISO 8859-13 (Alfabeto latino nº 7)
LATIN8	ISO 8859-14 (Alfabeto latino nº 8)

Nome	Descrição
LATIN9	ISO 8859-15 (Alfabeto latino nº 9)
LATIN10	ISO 8859-16/ASRO SR 14111 (Alfabeto latino nº 10)
ISO_8859_5	ISO 8859-5/ECMA 113 (Latino/Cirílico)
ISO_8859_6	ISO 8859-6/ECMA 114 (Latino/Arábico)
ISO_8859_7	ISO 8859-7/ECMA 118 (Latino/Grego)
ISO_8859_8	ISO 8859-8/ECMA 121 (Latino/Hebreu)
KOI8	KOI8-R(U)
ALT	Windows CP866
WIN874	Windows CP874 (Thai)
WIN1250	Windows CP1250
WIN	Windows CP1251
WIN1256	Windows CP1256 (Arábico)
TCVN	TCVN-5712/Windows CP1258 (Vietnamês)
Notas: a. UTF8 é a forma de codificação de caracteres especificada na ISO/IEC 10646-1, Anexo D, na qual cada caractere é codificado de um a quatro octetos. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)	

**Importante:** Por engano, antes do PostgreSQL 7.2 `LATIN5` significava ISO 8859-5. A partir da versão 7.2 `LATIN5` passou a significar ISO 8859-9. Caso exista um banco de dados `LATIN5` criado pela versão 7.1 ou anterior, e for desejado migrar para a versão 7.2 ou posterior, deve haver cuidado com relação a esta modificação.

Nem todas as APIs suportam todos os conjuntos de caracteres listados. Por exemplo, o driver de JDBC do PostgreSQL não aceita `MULE_INTERNAL`, `LATIN6`, `LATIN8` e `LATIN10`.

### 20.2.2. Definição do conjunto de caracteres

O utilitário `initdb` define o conjunto de caracteres padrão para o agrupamento de bancos de dados do PostgreSQL. Por exemplo,

```
initdb -E EUC_JP
```

define o conjunto de caracteres padrão (codificação) como `EUC_JP` (Código Unix Estendido para Japonês). Pode ser utilizado `--encoding` em vez de `-E`, se for preferido digitar a forma mais longa das opções. Se não for fornecida nem a opção `-E` nem a opção `--encoding`, é utilizado `SQL_ASCII`.

Pode ser criado um banco de dados com um conjunto de caracteres diferente:

```
createdb -E EUC_KR coreano
```

Este comando cria um banco de dados chamado `coreano` que utiliza o conjunto de caracteres `EUC_KR`. Outra forma de se fazer é através do comando SQL:

```
CREATE DATABASE coreano WITH ENCODING 'EUC_KR';
```

A codificação usada no banco de dados é armazenada no catálogo do sistema `pg_database`. Pode ser vista utilizando a opção `-l` ou o comando `\l` do `psql`.

```
$ psql -l
```

Lista de bancos de dados

Banco de Dados	Dono	Codificação
-----+-----+-----		
euc_cn	t-ishii	EUC_CN
euc_jp	t-ishii	EUC_JP
euc_kr	t-ishii	EUC_KR
euc_tw	t-ishii	EUC_TW
mule_internal	t-ishii	MULE_INTERNAL
regression	t-ishii	SQL_ASCII
templatel	t-ishii	EUC_JP
test	t-ishii	EUC_JP
unicode	t-ishii	UNICODE

(9 linhas)

**Importante:** Embora possa ser especificado para o banco de dados qualquer codificação desejada, não é razoável escolher uma codificação que não é a esperada para o idioma escolhido. As definições de `LC_COLLATE` e `LC_CTYPE` implicam em uma determinada codificação e, por isso, é possível que as operações dependentes do idioma (como a classificação) interpretem de forma errada os dados que estiverem em uma codificação incompatível.

Uma vez que as definições de idioma são congeladas pelo utilitário `initdb`, a flexibilidade aparente de utilizar codificações diferentes em bancos de dados diferentes é mais teórica do que real. É provável que estes mecanismos sejam revistos em uma versão futura do PostgreSQL.

Uma maneira de utilizar várias configurações com segurança é definir o idioma como `C` ou `POSIX` ao executar o `initdb` acabando, assim, com qualquer preocupação real com relação a idioma.

### 20.2.3. Conversão automática do conjunto de caracteres entre cliente e servidor

O PostgreSQL suporta a conversão automática de conjuntos de caracteres entre o cliente e o servidor, para determinados conjuntos de caracteres. A informação de conversão é armazenada no catálogo do sistema `pg_conversion`. Podem ser criadas novas conversões utilizando o comando `CREATE CONVERSION`. O PostgreSQL possui algumas conversões pré-definidas, conforme mostrado na Tabela 20-2.

**Tabela 20-2. Conversões de conjuntos de caracteres cliente/servidor**

Conjunto de caracteres do servidor	Conjuntos de caracteres do cliente aceitos
SQL_ASCII	SQL_ASCII, UNICODE, MULE_INTERNAL
EUC_JP	EUC_JP, SJIS, UNICODE, MULE_INTERNAL
EUC_CN	EUC_CN, UNICODE, MULE_INTERNAL
EUC_KR	EUC_KR, UNICODE, MULE_INTERNAL
JOHAB	JOHAB, UNICODE
EUC_TW	EUC_TW, BIG5, UNICODE, MULE_INTERNAL
LATIN1	LATIN1, UNICODE, MULE_INTERNAL
LATIN2	LATIN2, WIN1250, UNICODE, MULE_INTERNAL
LATIN3	LATIN3, UNICODE, MULE_INTERNAL
LATIN4	LATIN4, UNICODE, MULE_INTERNAL
LATIN5	LATIN5, UNICODE
LATIN6	LATIN6, UNICODE, MULE_INTERNAL
LATIN7	LATIN7, UNICODE, MULE_INTERNAL
LATIN8	LATIN8, UNICODE, MULE_INTERNAL

Conjunto de caracteres do servidor	Conjuntos de caracteres do cliente aceitos
LATIN9	LATIN9, UNICODE, MULE_INTERNAL
LATIN10	LATIN10, UNICODE, MULE_INTERNAL
ISO_8859_5	ISO_8859_5, UNICODE, MULE_INTERNAL, WIN, ALT, KOI8
ISO_8859_6	ISO_8859_6, UNICODE
ISO_8859_7	ISO_8859_7, UNICODE
ISO_8859_8	ISO_8859_8, UNICODE
UNICODE	EUC_JP, SJIS, EUC_KR, UHC, JOHAB, EUC_CN, GBK, EUC_TW, BIG5, LATIN1 até LATIN10, ISO_8859_5, ISO_8859_6, ISO_8859_7, ISO_8859_8, WIN, ALT, KOI8, WIN1256, TCVN, WIN874, GB18030, WIN1250
MULE_INTERNAL	EUC_JP, SJIS, EUC_KR, EUC_CN, EUC_TW, BIG5, LATIN1 até LATIN5, WIN, ALT, WIN1250, BIG5, ISO_8859_5, KOI8
KOI8	ISO_8859_5, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL
ALT	ISO_8859_5, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL
WIN874	WIN874, UNICODE
WIN1250	LATIN2, WIN1250, UNICODE, MULE_INTERNAL
WIN	ISO_8859_5, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL
WIN1256	WIN1256, UNICODE
TCVN	TCVN, UNICODE

Para habilitar a conversão automática entre os conjuntos de caracteres, deve ser informado ao PostgreSQL o conjunto de caracteres (codificação) que se deseja utilizar no cliente. Existem diversas maneiras de fazer:

- Utilizando o comando `\encoding` no `psql`. O `\encoding` permite mudar a codificação do cliente dinamicamente. Por exemplo, para mudar a codificação para `SJIS` deve ser executado:

```
\encoding SJIS
```

- Utilizando as funções da biblioteca `libpq`. O comando `\encoding` na verdade chama `PQsetClientEncoding()` para esta finalidade:

```
int PQsetClientEncoding(PGconn *conexão, const char *codificação);
```

Onde *conexão* é a conexão com o servidor, e *codificação* é a codificação que se deseja utilizar. Se for bem-sucedida a definição da codificação pela função, esta retorna 0, senão retorna -1. A codificação corrente para a conexão pode ser obtida utilizando:

```
int PQclientEncoding(const PGconn *conexão);
```

Deve ser observado que é retornado o ID da codificação, e não uma cadeia de caracteres simbólica como `EUC_JP`. Para converter o ID da codificação no nome da codificação deve ser utilizado:

```
char *pg_encoding_to_char(int id_codificação);
```

- Utilizando o comando `SET client_encoding TO`. A definição da codificação do cliente pode ser feita através do comando SQL:

```
SET CLIENT_ENCODING TO 'valor';
```

Também pode ser utilizada a sintaxe `SET NAMES` do padrão SQL para esta finalidade:

```
SET NAMES 'valor';
```

Para consultar a codificação corrente do cliente:

```
SHOW client_encoding;
```

Para voltar à codificação padrão:

```
RESET client_encoding;
```

- Utilizando a variável de ambiente `PGCLIENTENCODING`. Se a variável de ambiente `PGCLIENTENCODING` estiver definida no ambiente do cliente, esta codificação de cliente é selecionada automaticamente quando é feita a conexão com o servidor (Pode ser mudada depois utilizando um dos métodos mencionados acima).
- Utilizando a variável de configuração `client_encoding`. Se a variável `client_encoding` estiver definida, esta codificação do cliente é selecionada automaticamente quando é feita a conexão com o servidor (Pode ser mudada depois utilizando um dos métodos mencionados acima).

Se não for possível converter um determinado caractere — suponha que seja escolhido `EUC_JP` para o servidor e `LATIN1` para o cliente, então alguns caracteres Japoneses não poderão ser convertidos em `LATIN1` — este caractere será transformado nos valores hexadecimais de seus bytes entre parênteses como, por exemplo, `(826C)`.

### 20.2.4. Leitura adicional

Abaixo estão mostrados bons lugares para começar a aprender os vários tipos de sistema de codificação.

<ftp://ftp.ora.com/pub/examples/nutshell/ujip/doc/cjk.inf>

Estão mostradas na seção 3.2 explicações detalhadas sobre `EUC_JP`, `EUC_CN`, `EUC_KR` e `EUC_TW`.

<http://www.unicode.org/>

O sítio na Web do Consórcio Unicode

RFC 2044

Onde o UTF-8 é definido.

## Notas

1. O comando `locale -a` executado no Fedora Core 3 mostrou a seguinte relação: `aa_DJ`, `aa_DJ.iso88591`, `aa_ER`, `aa_ER@saaho`, `aa_ER.utf8`, `aa_ER.utf8@saaho`, `aa_ET`, `aa_ET.utf8`, `af_ZA`, `af_ZA.iso88591`, ... , `portuguese`, `POSIX`, `pt_BR`, `pt_BR.iso88591`, `pt_BR.utf8`, `pt_PT`, `pt_PT@euro`, `pt_PT.iso88591`, `pt_PT.iso885915@euro`, `pt_PT.utf8`, ... , `zu_ZA`, `zu_ZA.iso88591`, `zu_ZA.utf8`. (N. do T.)

# Capítulo 21. Rotinas de manutenção do banco de dados

Existem algumas poucas tarefas de manutenção que precisam ser realizadas regularmente para manter o servidor PostgreSQL funcionando sem problemas. As tarefas mostradas neste capítulo são repetitivas por natureza, podendo ser facilmente automatizadas utilizando as ferramentas padrão do Unix como os scripts do cron. É responsabilidade do administrador do banco de dados instalar os scripts apropriados, e verificar se a execução está sendo bem-sucedida.

Uma tarefa de manutenção óbvia é a geração das cópias de segurança dos dados em períodos regulares. Sem uma cópia de segurança recente, não há como fazer a recuperação após um desastre (falha de disco, incêndio, remoção por engano de uma tabela crítica, etc.). Os mecanismos de cópia de segurança e restauração disponíveis no PostgreSQL estão descritos de forma abrangente no Capítulo 22.

Outra tarefa de manutenção importante é a limpeza periódica do banco de dados. Esta atividade está descrita na Seção 21.1.

Outro item que pode precisar de atenção periódica é o gerenciamento do arquivo de registro (`log`), conforme mostrado na Seção 21.3.

O PostgreSQL necessita de pouca manutenção se comparado a outros sistemas gerenciadores de bancos de dados. Apesar disso, a devida atenção a estas tarefas garante bem mais que uma experiência agradável e produtiva do sistema.

## 21.1. Rotina de Limpeza

O comando `VACUUM` do PostgreSQL deve ser executado regularmente por diversos motivos:

1. Para recuperar o espaço em disco ocupado pelas linhas atualizadas e removidas.
2. Para atualizar as estatísticas dos dados utilizadas pelo planejador de comandos do PostgreSQL.
3. Para proteger contra perda de dados muito antigos devido ao *recomeço do ID de transação*.

A frequência e a abrangência das operações de `VACUUM`, realizadas devido aos motivos acima, variam dependendo das necessidades da instalação. Portanto, os administradores de banco de dados devem compreender estas questões e desenvolver uma estratégia de manutenção apropriada. Esta seção se concentra em explicar as questões de alto nível; para obter detalhes sobre a sintaxe do comando e outras informações deve ser consultada a página de referência do comando `VACUUM`.

A partir do PostgreSQL 7.2 a forma padrão do comando `VACUUM` pode executar em paralelo com as operações normais do banco de dados (seleções, inserções, atualizações, exclusões, mas sem modificação de definição de tabela). Portanto, a rotina de limpeza não é mais tão impactante como era nas versões anteriores, não sendo mais tão crítico tentar agendá-la para as horas do dia com baixa utilização.

A partir do PostgreSQL 8.0 passaram a existir parâmetros de configuração que podem ser ajustados para reduzir ainda mais o impacto da limpeza em segundo plano. Consulte a Seção 16.4.3.4.

### 21.1.1. Recuperação do espaço em disco

Na operação normal do PostgreSQL, um comando `UPDATE` ou `DELETE` em uma linha não remove imediatamente a versão antiga da linha. Esta abordagem é necessária para obter os benefícios do controle de simultaneidade multi-versão (consulte o Capítulo 12): a versão da linha não pode ser removida enquanto houver possibilidade de ser acessada por outras transações. Mas no final, uma versão de linha desatualizada ou excluída não terá mais interesse para nenhuma transação. O espaço ocupado deve ser recuperado para ser reutilizado pelas novas linhas, evitando um crescimento sem fim da necessidade de espaço em disco. Isto é feito executando o comando `VACUUM`.

Obviamente, uma tabela que recebe atualizações ou exclusões frequentes necessita ser limpa com mais frequência que uma tabela que é atualizada raramente. Pode ser útil configurar tarefas periódicas no aplicativo cron para limpar apenas determinadas tabelas, omitindo as tabelas sabidamente pouco modificadas. Provavelmente, este procedimento é útil apenas quando há tanto tabelas muito atualizadas quanto tabelas raramente atualizadas — o custo adicional para limpar uma tabela pequena não é suficiente para valer a pena se preocupar com isto.

Existem duas variações do comando `VACUUM`. A primeira forma, conhecida como “limpeza preguiçosa” (`lazy vacuum`), ou simplesmente `VACUUM`, marca os dados expirados das tabelas para reutilização posterior; *não* tenta recuperar o espaço

utilizado pelos dados expirados imediatamente. Portanto, o arquivo da tabela não é encurtado, e o espaço não utilizado no arquivo não é devolvido ao sistema operacional. Esta variante do `VACUUM` pode ser executada simultaneamente com as operações normais do banco de dados.

A segunda forma é o comando `VACUUM FULL`. Esta forma utiliza um algoritmo mais agressivo para recuperar o espaço consumido pelas versões de linha expiradas. Todo espaço liberado pelo `VACUUM FULL` é imediatamente devolvido ao sistema operacional. Infelizmente, esta variante do comando `VACUUM` obtém um bloqueio exclusivo de cada tabela enquanto esta é processada pelo comando `VACUUM FULL`. Portanto, a utilização freqüente do comando `VACUUM FULL` pode ter um efeito extremamente negativo sobre o desempenho dos comandos simultâneos no banco de dados.

A forma padrão do comando `VACUUM` é melhor empregada com o objetivo de manter o nível de utilização de espaço em disco razoavelmente estável. Se for necessário devolver espaço em disco para o sistema operacional, pode ser utilizado o comando `VACUUM FULL` — mas qual é a vantagem de liberar espaço em disco que deverá ser alocado novamente em breve? Execuções do comando `VACUUM` padrão com freqüência moderada é uma abordagem melhor que a execução do comando `VACUUM FULL` com baixa freqüência, para manutenção de tabelas muito atualizadas.

A prática recomendada para a maioria das instalações é agendar o comando `VACUUM` para todo o banco de dados uma vez por dia em horário de pouca utilização, suplementado por limpezas mais freqüentes das tabelas muito atualizadas se for necessário (Algumas instalações com taxas muito alta de modificação dos dados executam o comando `VACUUM` em tabelas muito atualizadas uma vez a cada poucos minutos). Havendo vários bancos de dados em um agrupamento, não deve ser esquecido de limpar cada um deles; o programa `vacuumdb` pode ser útil.

**Dica:** O programa `contrib/pg_autovacuum` pode ser útil para automatizar operações de limpeza com alta freqüência.

O comando `VACUUM FULL` é recomendado para os casos onde se sabe que foi excluída a maior parte das linhas da tabela e, portanto, o tamanho estável da tabela pode ser reduzido substancialmente pela abordagem mais agressiva do comando `VACUUM FULL`. Deve ser utilizado o `VACUUM` simples, e não o `VACUUM FULL`, para recuperação rotineira de espaço.

Havendo uma tabela cujo conteúdo é excluído periodicamente, deve ser considerado executar o comando `TRUNCATE` em vez de utilizar `DELETE` seguido por `VACUUM`. O comando `TRUNCATE` remove todo o conteúdo da tabela imediatamente, não sendo necessário executar o comando `VACUUM` ou `VACUUM FULL` em seguida para recuperar o espaço que não está mais sendo utilizado.

### 21.1.2. Atualização das estatísticas do planejador

O planejador de comandos do PostgreSQL depende das informações estatísticas sobre o conteúdo das tabelas para poder gerar bons planos para os comandos. Estas estatísticas são coletadas pelo comando `ANALYZE`, que pode ser chamado por si próprio ou como um passo opcional do comando `VACUUM`. É importante que as estatísticas estejam razoavelmente precisas, senão o desempenho do banco de dados poderá ser degradado por planos mal escolhidos.

Assim como a execução do comando `VACUUM` para recuperar espaço, atualizações freqüentes das estatísticas são mais úteis para tabelas muito atualizadas que para tabelas raramente atualizadas. Porém, mesmo nas tabelas muito atualizadas, pode não ser necessário atualizar as estatísticas, se a distribuição dos dados não mudar muito. Uma regra empírica simples é pensar sobre quanto os valores mínimo e máximo das colunas da tabela mudam. Por exemplo, uma coluna `timestamp` contendo a data e hora da atualização da linha terá um valor máximo aumentando continuamente na medida em que forem atualizadas ou adicionadas linhas à tabela; este tipo de coluna, provavelmente, precisa de atualizações mais freqüentes das estatísticas do que, digamos, uma coluna contendo URLs de páginas acessadas em sítios da Web. A coluna URL pode ser modificada com a mesma freqüência, mas a distribuição estatística de seus valores provavelmente muda de forma relativamente lenta.

É possível executar o comando `ANALYZE` em tabelas específicas, e mesmo em colunas específicas da tabela. Portanto, existe flexibilidade para atualizar algumas estatísticas com mais freqüência que outras se for requerido pelo aplicativo. Entretanto, na prática a utilidade desta funcionalidade é duvidosa. A partir do PostgreSQL 7.2 o comando `ANALYZE` se tornou uma operação bem rápida, mesmo em tabelas grandes, porque utiliza uma amostra aleatória das linhas da tabela, em vez de ler todas as linhas da tabela. Portanto, provavelmente é mais simples executá-lo para todo o banco de dados na freqüência desejada.

**Dica:** Embora possa não ser muito produtivo aumentar a freqüência de execução do comando `ANALYZE` por coluna, pode valer a pena fazer ajustes por coluna do nível de detalhe das estatísticas coletadas pelo comando `ANALYZE`. Colunas que são muito utilizadas em cláusula `WHERE`, e que contém uma distribuição de dados muito irregular, podem

requerer um histograma dos dados com granulação mais fina que as demais colunas. Consulte o comando `ALTER TABLE SET STATISTICS`.

A prática recomendada, para a maioria das instalações, é agendar a execução do comando `ANALYZE` para todo o banco de dados uma vez por dia, em horário de pouca utilização; é útil sua combinação com a execução do comando `VACUUM` todas as noites. Entretanto, nas instalações onde as estatísticas das tabelas mudam de forma relativamente lenta, pode-se considerar que esta frequência seja demasiada, e que a execução do comando `ANALYZE` com uma frequência mais baixa seja suficiente.

### 21.1.3. Prevenção de falhas devido ao reinício do ID de transação

A semântica de transação do MVCC do PostgreSQL depende de poder comparar números identificadores de transação (XID): uma versão de linha com XID de inserção maior que o XID da transação corrente está “no futuro”, não devendo ser enxergada pela transação corrente. Como os IDs de transação possuem tamanho limitado (32 bits quando esta documentação foi escrita), um agrupamento em funcionamento por um longo período de tempo (mais de 4 bilhões de transações) sofre um *reinício do ID de transação*: o contador do XID volta a zero e, de repente, as transações que estavam no passado parecem estar no futuro — significando que suas saídas se tornam invisíveis. Em resumo, uma perda de dados catastrófica (Na verdade os dados ainda estão lá, mas isto não serve de consolo se não é possível acessá-los).

Antes do PostgreSQL 7.2 a única defesa contra o reinício do XID era executar novamente o `initdb` pelo menos a cada 4 bilhões de transações. É claro que não era muito satisfatório para instalações com alto tráfego e, por isso, foi concebida uma solução melhor. A nova abordagem permite o servidor permanecer ativo indefinidamente, sem executar o `initdb` ou qualquer forma de reinício. O preço é a necessidade desta manutenção: *todas as tabelas do banco de dados devem ser VACUUM-nizadas pelo menos uma vez a cada um bilhão de transações*.

Na prática este não é um requisito oneroso, mas uma vez que a consequência de não respeitá-lo pode ser a perda total dos dados (e não apenas desperdício de espaço em disco ou degradação do desempenho), foram introduzidos alguns dispositivos especiais para ajudar os administradores de banco de dados acompanharem o tempo decorrido desde que o comando `VACUUM` foi executado pela última vez. O restante desta seção fornece os detalhes.

A nova abordagem para comparação de XID faz distinção de dois XIDs especiais, os de número 1 e 2 (`BootstrapXID` e `FrozenXID`). Estes dois XIDs são sempre considerados mais antigos que qualquer XID normal. Os XIDs normais (àqueles maiores que 2) são comparados utilizando a aritmética de módulo-2<sup>31</sup>. Isto significa que para todo XID normal existem dois bilhões de XIDs que são “mais antigos” e dois bilhões que são “mais novos”; outra maneira disto ser dito é que o espaço do XID normal é circular, sem ponto de término. Portanto, ao ser criada uma versão de linha com um determinado XID normal, esta versão de linha vai parecer estar “no passado” para as próximas dois bilhões de transações, não importando de qual XID normal está se falando. Se a versão da linha ainda existir após mais de dois bilhões de transações, de repente vai parecer estar no futuro. Para evitar perda de dados, deve ser atribuído o XID `FrozenXID` para as versões antigas das linhas algum tempo antes de atingirem a marca “antiga-dois-bilhões-de-transações”. Uma vez que tenha sido atribuído este XID especial, vai parecer estar “no passado” para todas as transações normais, a despeito dos problemas de reinício, e esta versão de linha será válida até ser excluída, não importando quanto demore. Esta reatribuição de XID é tratada pelo comando `VACUUM`.

A maneira normal de agir do comando `VACUUM` é atribuir o `FrozenXID` para toda versão de linha que possua um XID normal antigo que esteja mais de um bilhão de transações no passado. Esta política preserva o XID original de inserção até que, provavelmente, não seja mais de interesse (Na verdade, a maioria das versões de linha provavelmente vivem e morrem sem que jamais tenham sido “congeladas”). Com esta política, o intervalo máximo seguro entre execuções do comando `VACUUM` em qualquer tabela é de exatamente um bilhão de transações: se for esperado mais tempo, é possível que uma versão de linha que da última vez não era antiga o suficiente para ser congelada, agora esteja antiga mais de dois bilhões de transações, e tenha passado para o futuro — ou seja, foi perdida (É claro que vai reaparecer após outros dois bilhões de transações, mas isto não ajuda).

Uma vez que a execução periódica do comando `VACUUM` é necessária devido aos motivos descritos anteriormente, é pouco provável que alguma tabela não tenha sido limpa pelo tempo de um bilhão de transações. Para ajudar os administradores a garantir que esta restrição é obedecida, o comando `VACUUM` armazena estatísticas sobre ID de transação na tabela do sistema `pg_database`. Em particular, ao término da operação de limpeza de todo o banco de dados (ou seja, o comando `VACUUM` sem especificação de um nome de tabela), é atualizada a coluna `datfrozenxid` da linha do banco de dados na tabela `pg_database`. O valor armazenado neste campo é o XID do ponto de corte de congelamento utilizado pelo comando `VACUUM`. Há garantia que, neste banco de dados todos, os XIDs mais antigos que este XID de ponto de corte foram substituídos pelo `FrozenXID`. Uma maneira conveniente de examinar esta informação é executar a consulta:



```
SELECT datname, age(datfrozenxid) FROM pg_database;
```

A coluna `age` (idade) mede o número de transações desde o XID de corte de congelamento até o XID da transação corrente.

Com a política de congelamento padrão, a coluna `age` começa em um bilhão para um banco de dados onde o comando `VACUUM` acabou de ser executado. Quando `age` se aproxima de dois bilhões, deve ser executado novamente o comando `VACUUM` no banco de dados para evitar o risco da falha devido ao reinício. A prática recomendada é executar o comando `VACUUM` em todos os bancos de dados pelo menos uma vez a cada meio bilhão (500 milhões) de transações, para que se tenha uma ampla margem de segurança. Para ajudar a obedecer esta regra, cada execução do comando `VACUUM` de todo o banco de dados emite, automaticamente, uma advertência caso haja alguma entrada em `pg_database` mostrando uma idade de mais 1,5 bilhões de transações. Por exemplo:

```
play=# VACUUM;
WARNING:  some databases have not been vacuumed in 1613770184 transactions
HINT:     Better vacuum them within 533713463 transactions, or you may have a wraparound failure.
VACUUM
```

O comando `VACUUM` com a opção `FREEZE` utiliza uma política de congelamento mais agressiva: as versões das linhas são congeladas se forem antigas o suficiente para serem consideradas boas por todas as transações em aberto. Em particular, se o comando `VACUUM FREEZE` for executado em um banco de dados não utilizado de outra forma, é garantido que *todas* as versões de linha neste banco de dados serão congeladas. Portanto, enquanto o banco de dados não for modificado de forma alguma, não será necessário executar o comando `VACUUM` para evitar o problema de reinício do ID de transação. Esta técnica é utilizada pelo `initdb` para preparar o banco de dados `template0`. Também deve ser utilizada para preparar todos os bancos de dados criados pelo usuário a serem marcados com `dataallowconn = false` em `pg_database`, uma vez que não há nenhuma maneira conveniente de executar o comando `VACUUM` em um banco de dados em que não se pode conectar. Deve ser observado que a mensagem de advertência automática do comando `VACUUM` sobre bancos de dados não limpos ignoram as entradas de `pg_database` com `dataallowconn = false`, para evitar emitir falsas advertências sobre estes bancos de dados; portanto, é responsabilidade de quem o faz garantir que estes bancos de dados sejam congelados corretamente.

### Atenção

Para garantir a segurança contra reinício de transação é necessário limpar *todas* as tabelas, inclusive os catálogos do sistema, em *todos* os bancos de dados, pelo menos uma vez a cada bilhão de transações. Já foi visto perda de dados causadas por pessoas que decidiram que bastava limpar suas próprias tabelas de usuário ativas. Isto vai parecer que funciona bem, mas só por algum tempo.

## 21.2. Rotina de reindexação

Em algumas situações vale a pena reconstruir índices periodicamente utilizando o comando `REINDEX` (Existe, também, o aplicativo `contrib/reindexdb` que pode reindexar todo o banco de dados). Entretanto, o PostgreSQL 7.4 reduziu de forma substancial a necessidade desta atividade se comparado às versões anteriores.

## 21.3. Manutenção do arquivo de registro

É uma boa idéia salvar a saída do registro (`log`) do servidor de banco de dados em algum lugar, em vez de simplesmente direcionar para `/dev/null`. A saída do registro é valiosa para fazer diagnóstico de problemas. Entretanto, a saída do registro tende a se tornar volumosa (especialmente nos níveis de depuração altos), e não será desejado salvá-la indefinidamente. É necessário fazer a “rotação” dos arquivos de registro, para que sejam iniciados novos arquivos, e os arquivos antigos sejam removidos periodicamente.

Se for simplesmente direcionada a saída `stderr` do `postmaster` para um arquivo, haverá uma saída de registro, mas a única maneira de truncar o arquivo de registro será parando e reiniciando o `postmaster`, o que pode ser adequado em um ambiente de desenvolvimento, mas poucos ambientes de produção vão considerar este comportamento aceitável.

Uma abordagem melhor é enviar a saída `stderr` do `postmaster` para algum tipo de programa de rotação de registro. Existe um programa nativo de rotação de registro, que pode ser utilizado definindo o parâmetro de configuração

`redirect_stderr` como `true` no arquivo `postgresql.conf`. Os parâmetros de controle para este programa estão descritos na Seção 16.4.6.1.

Como alternativa, pode-se preferir utilizar um programa de rotação de registro externo, se já houver um sendo usado por outro programa servidor. Por exemplo, a ferramenta `rotatelog`s incluída na distribuição do Apache pode ser utilizada pelo PostgreSQL. Para isto ser feito, a saída `stderr` do `postmaster` deve ser canalizada (`pipe`) para o programa desejado. Se o servidor for inicializado pelo `pg_ctl`, então a saída `stderr` do `postmaster` já estará redirecionada para `stdout` e, portanto, somente será necessário utilizar o operador `pipe` (`|`), como, por exemplo:

```
pg_ctl start | rotatelog /var/log/pgsql_log 86400
```

Outra abordagem para gerenciar a saída do registro, apropriada para ambientes de produção, é enviar a saída para `syslog` e deixar que algum programa cuide da rotação do registro. Para que isto seja feito, deve ser definido no arquivo `postgresql.conf` o parâmetro de configuração `log_destination` como `syslog` (para registrar apenas em `syslog`). Então, pode ser enviado o sinal `SIGHUP` para o processo (`daemon`) `syslogd` para que este realize a reinicialização (fechar todos os arquivos abertos, ler novamente o arquivo de configuração, e iniciar a facilidade `syslog` novamente).<sup>1 2</sup> Se for desejado automatizar a rotação do registro, pode ser configurado o programa `logrotate` para trabalhar com os arquivos de registro do `syslog`.<sup>3</sup>

Entretanto, em muitos sistemas o `syslog` não é muito confiável, particularmente com mensagens de registro grandes; pode truncar ou remover as mensagens justamente quando forem mais necessárias. Também, no Linux, o `syslog` sincroniza todas as mensagens com o disco, ocasionando um desempenho medíocre (Pode ser utilizado um hífen (`-`) no início do nome do arquivo, no arquivo de configuração do `syslog`, para desabilitar este comportamento).

Deve ser observado que todas as soluções descritas acima têm o cuidado de iniciar novos arquivos de registro a intervalos configuráveis, mas não tratam dos arquivos de registro antigos, que não são mais de interesse. Provavelmente, será desejado definir um script executado periodicamente para remover os arquivos de registro antigos. Outra possibilidade é configurar o programa de rotação para que os arquivos de registro antigos sejam sobrescritos ciclicamente.

## Notas

1. Fedora Core 3 — `kill -s SIGHUP `cat /var/run/syslogd.pid`` (N. do T.)
2. `syslogd` é um mecanismo que permite que qualquer comando registre mensagens na console do sistema e/ou em um arquivo. O `daemon` `syslogd` recebe as mensagens dos comandos e envia para o destino descrito no arquivo de configuração ( `/etc/syslog.conf` ). O `syslogd` `daemon` lê a configuração quando é inicializado e quando recebe um signal de hangup ( `kill -HUP processo` ). Dicas-L (<http://www.dicas-l.unicamp.br/cursos/seguranca/seguranca-150.html>) (N. do T.)
3. O `logrotate` foi projetado para facilitar a administração dos sistemas que geram um número grande de arquivos de registro. Permite a rotação automática, compressão, remoção e envio por correio eletrônico dos arquivos de registro. Cada arquivo de registro pode ser tratado diariamente, semanalmente, mensalmente ou quando fica muito grande. (N. do T.)

## Capítulo 22. Criação e restauração de cópias de segurança

Como tudo que contém dados importantes, devem ser feitas cópias de segurança dos bancos de dados do PostgreSQL regularmente. Embora o procedimento seja essencialmente simples, é importante possuir uma compreensão básica das técnicas e princípios subjacentes.

Existem três abordagens fundamentalmente diferentes para fazer cópia de segurança dos dados do PostgreSQL:

- Método SQL-dump
- Cópia de segurança no nível de sistema de arquivos
- Cópia de segurança em-linha

Cada uma tem seus próprios pontos fortes e pontos fracos.

### 22.1. Método SQL-dump

A idéia por trás do Método SQL-dump é gerar um arquivo texto contendo comandos SQL que, ao serem processados pelo servidor, recriam o banco de dados no mesmo estado em que este se encontrava quando o arquivo foi gerado. O PostgreSQL disponibiliza o programa utilitário `pg_dump` para esta finalidade. A forma básica de utilização deste programa é:

```
pg_dump nome_do_banco_de_dados > arquivo_de_saída
```

Conforme pode ser visto, o programa `pg_dump` escreve o seu resultado na saída padrão. Será visto abaixo como isto pode ser útil.

O `pg_dump` é um aplicativo cliente normal do PostgreSQL (embora seja particularmente astuta). Isto significa que o procedimento de cópia de segurança pode ser realizado a partir de qualquer hospedeiro remoto que possua acesso ao banco de dados. Porém, deve ser lembrado que o `pg_dump` não opera com permissão especial. Em particular, é necessário possuir acesso de leitura a todas as tabelas que se deseja fazer cópia de segurança. Portanto, na prática, quase sempre é necessário ser um superusuário do banco de dados.

Para especificar qual servidor de banco de dados o `pg_dump` deve se conectar, devem ser utilizadas as opções de linha de comando `-h hospedeiro` e `-p porta`. O hospedeiro padrão é o hospedeiro local, ou o que estiver especificado na variável de ambiente `PGHOST`. De maneira semelhante, a porta padrão é indicada pela variável de ambiente `PGPORT` ou, na falta desta, pelo padrão de compilação (Por conveniência, o servidor normalmente é compilado usando o mesmo padrão).

Assim como qualquer outro aplicativo cliente do PostgreSQL, o `pg_dump` se conecta por padrão ao banco de dados cujo nome é igual ao nome do usuário corrente do sistema operacional. Para que seja outro, deve ser especificada a opção `-U`, ou definida a variável de ambiente `PGUSER`. Não deve ser esquecido que as conexões do `pg_dump` estão sujeitas aos mecanismos normais de autenticação de cliente (conforme descritos no Capítulo 19).

As cópias de segurança criadas pelo `pg_dump` são consistentes internamente, ou seja, as atualizações feitas no banco de dados enquanto o `pg_dump` está executando não estão presentes na cópia de segurança. O `pg_dump` não bloqueia outras operações no banco de dados enquanto está executando (Exceto as operações que necessitam operar com modo de bloqueio exclusivo, como o `VACUUM FULL`).

**Importante:** Quando o esquema do banco de dados é dependente dos OIDs (como chaves estrangeiras, por exemplo) deve-se instruir o `pg_dump` para que também inclua os OIDs. Para que isto seja feito, deve ser utilizada a opção de linha de comando `-o`. Também, não são feitas cópias de segurança dos “objetos grandes” por padrão. Se forem utilizados objetos grandes, deve ser consultada a página de referência do programa `pg_dump`.

#### 22.1.1. Restauração da cópia de segurança

Os arquivos texto criados pelo programa `pg_dump` são feitos para serem lidos pelo programa `psql`. A forma geral do comando para restaurar uma cópia de segurança é:

```
psql nome_do_banco_de_dados < arquivo_de_entrada
```

onde o `arquivo_de_entrada` é o que foi utilizado como `arquivo_de_saída` pelo programa `pg_dump`. O banco de dados `nome_do_banco_de_dados` não será criado por este comando, devendo ser criado a partir de `template0` antes de executar o `psql` (por exemplo, usando `createdb -T template0 nome_do_banco_de_dados`). O `psql` possui opções semelhantes às do `pg_dump` para controlar a identificação do servidor de banco de dados e o nome do usuário. Para obter informações adicionais deve ser consultada a página de referência do programa `psql`.

Antes de começar a executar a restauração, não basta existir o banco de dados de destino. Devem existir, também, todos os usuários que possuam objetos ou concessões para os objetos contidos na cópia de segurança do banco de dados. Caso estes usuários não existam, a restauração não será capaz de recriar os objetos com os mesmos donos e concessões originais (Algumas vezes é o que se deseja, mas geralmente não é).

Uma vez feita a restauração, é sensato executar o comando `ANALYZE` em cada um dos bancos de dados, para que o otimizador possua estatísticas úteis. Uma forma fácil de se fazer é executando `vacuumdb -a -z` para efetuar o `VACUUM ANALYZE` de todos os bancos de dados; equivale a executar `VACUUM ANALYZE` manualmente.

A capacidade do `pg_dump` e do `psql` de escrever e ler de canais (`pipes`) torna possível replicar um banco de dados de um servidor para outro diretamente; por exemplo:

```
pg_dump -h hospedeiro1 nome_do_banco_de_dados | psql -h hospedeiro2 nome_do_banco_de_dados
```

**Importante:** As cópias de segurança produzidas pelo `pg_dump` são relativas a `template0`. Isto significa que todas as linguagens, procedimentos, etc. adicionados a `template1` também serão incluídos na cópia de segurança feita pelo `pg_dump`. Como resultado, se estiver sendo utilizado um banco de dados `template1` personalizado, ao ser feita a restauração deve ser criado um banco de dados vazio a partir de `template0`, conforme mostrado no exemplo acima.

Para obter conselhos sobre como carregar grande quantidade de dados no PostgreSQL de forma eficiente, deve ser consultado o Seção 13.4.

### 22.1.2. Utilização do `pg_dumpall`

O mecanismo mostrado acima não é cômodo nem apropriado para fazer a cópia de segurança de todo o agrupamento de bancos de dados. Por este motivo é fornecido o programa `pg_dumpall`. O `pg_dumpall` faz a cópia de segurança de todos os bancos de dados de um agrupamento, e também salva dados de todo o agrupamento, como os usuários e grupos. A forma básica de utilização deste programa é:

```
pg_dumpall > arquivo_de_saída
```

A cópia de segurança gerada pode ser restaurada pelo `psql` usando:

```
psql template1 < arquivo_de_entrada
```

(Na verdade, pode ser especificado qualquer nome de banco de dados existente para começar, mas se estiver sendo feita a restauração em um agrupamento vazio, então `template1` é a única escolha disponível). É sempre necessário possuir acesso de superusuário do banco de dados para fazer a restauração de uma cópia de segurança gerada pelo `pg_dumpall`, para poder restaurar as informações de usuário e de grupo.

### 22.1.3. Tratamento de bancos de dados grandes

Como o PostgreSQL permite a existência de tabelas maiores do que o tamanho máximo de arquivo do sistema operacional, pode ser problemático fazer a cópia de segurança de uma tabela como esta em um arquivo, uma vez que o arquivo resultante provavelmente terá um tamanho maior que o máximo permitido pelo sistema operacional. Como o `pg_dump` pode escrever na saída padrão, podem ser utilizadas ferramentas padrão do Unix para superar este possível problema.

**Utilização de cópias de segurança comprimidas.** Pode ser utilizado o programa de compressão favorito como, por exemplo, o `gzip`.

```
pg_dump nome_do_banco_de_dados | gzip > nome_do_arquivo.gz
```

Restaurar com

```
createdb nome_do_banco_de_dados
gunzip -c nome_do_arquivo.gz | psql nome_do_banco_de_dados
```

ou

```
cat nome_do_arquivo.gz | gunzip | psql nome_do_banco_de_dados
```

**Utilização do comando `split`.** O comando `split` permite dividir a saída em blocos de tamanho aceitável para o sistema de arquivos subjacente. Por exemplo, para fazer blocos de 1 megabyte:

```
pg_dump nome_do_banco_de_dados | split -b 1m - nome_do_arquivo
```

Restaurar com

```
createdb nome_do_banco_de_dados
cat nome_do_arquivo* | psql nome_do_banco_de_dados
```

**Utilização de formatos de cópia de segurança personalizados.** Se o PostgreSQL foi construído em um sistema com a biblioteca de compressão `zlib` instalada, o formato de cópia de segurança personalizado comprime os dados ao escrever o arquivo de saída. Produz cópias de segurança com tamanhos semelhantes às produzidas utilizando o `gzip`, mas tem a vantagem adicional de permitir a restauração seletiva das tabelas. O comando abaixo gera a cópia de segurança do banco de dados utilizando o formato de cópia de segurança personalizado (`custom dump format`):

```
pg_dump -Fc nome_do_banco_de_dados > nome_do_arquivo
```

O formato de cópia de segurança personalizado não é um script para o `psql`, devendo ser restaurado pelo `pg_restore`. Para obter detalhes devem ser vistas as páginas de referência do `pg_dump` e do `pg_restore`.

#### 22.1.4. Precauções

Por motivo de compatibilidade com as versões anteriores, o `pg_dump` não faz cópia de segurança dos objetos grandes por padrão. Para fazer cópia de segurança dos objetos grandes deve ser utilizado o formato de saída personalizado ou o formato `tar`, e utilizada a opção `-b` do `pg_dump`. Para obter detalhes deve ser consultada a página de referência do `pg_dump`. O diretório `contrib/pg_dumplo`, da árvore do código fonte do PostgreSQL, também contém um programa que pode ser utilizado para fazer cópias de segurança dos objetos grandes.

Por favor se familiarize com a página de referência do `pg_dump`.

## 22.2. Cópia de segurança no nível de sistema de arquivo

Uma estratégia alternativa para fazer cópia de segurança, é copiar diretamente os arquivos que o PostgreSQL usa para armazenar os dados dos bancos de dados. Na Seção 16.2 é explicado onde estes arquivos estão localizados, mas provavelmente você já os encontrou se está interessado neste método. Pode ser utilizada a forma preferida para fazer as cópias de segurança usuais dos arquivos do sistema como, por exemplo:

```
tar -cf copia_de_seguranca.tar /usr/local/pgsql/data
```

Entretanto, existem duas restrições que fazem com que este método seja impraticável ou, pelo menos, inferior ao `pg_dump`:

1. O servidor de banco de dados *deve* estar parado para que se possa obter uma cópia de segurança utilizável. Meias medidas, como impedir todas as conexões, não funcionam (principalmente porque o `tar`, e as ferramentas semelhantes, não capturam um instantâneo atômico do estado do sistema de arquivos em um determinado ponto no tempo). As informações sobre como parar o servidor podem ser encontradas na Seção 16.6. É desnecessário dizer que também é necessário parar o servidor antes de restaurar os dados.
2. Caso tenha se aprofundado nos detalhes da organização do sistema de arquivos do banco de dados, poderá estar tentado a fazer cópias de segurança ou restauração de apenas algumas determinadas tabelas ou bancos de dados a partir de seus respectivos arquivos ou diretórios. Isto *não* funciona, porque as informações contidas nestes arquivos possuem apenas meia verdade. A outra metade está nos arquivos de registro de efetivação `pg_clog/*`, que contém o status de efetivação de todas as transações. O arquivo da tabela somente possui utilidade com esta informação. É claro que também não é possível restaurar apenas uma tabela e seus dados associados em `pg_clog`, porque isto torna todas as outras tabelas do agrupamento de bancos de dados inúteis. Portanto, as cópias de segurança do sistema de arquivos somente funcionam para a restauração completa de todo o agrupamento de bancos de dados.

Uma abordagem alternativa para cópias de segurança do sistema de arquivos é fazer um “instantâneo consistente” do diretório de dados, se o sistema de arquivos possuir esta funcionalidade (e se há confiança que foi implementado de forma correta). O procedimento típico é tirar um “instantâneo congelado” (*frozen snapshot*) do volume que contém o banco de dados, depois copiar todo o diretório de dados (não apenas parte deste, veja acima) do instantâneo para uma unidade de cópia de segurança, e depois liberar o instantâneo congelado. Isto funciona mesmo com o banco de dados em operação. Entretanto, a cópia de segurança criada desta maneira salva os arquivos do banco de dados em um estado onde o servidor de banco de dados não foi parado de forma apropriada; portanto, quando o servidor de banco de dados é iniciado acessando um diretório restaurado a partir de uma cópia de segurança deste tipo, considera que o servidor caiu e refaz o registro do WAL. Isto não é um problema, mas deve-se estar atento a este fato (e certifique-se de incluir os arquivos do WAL na cópia de segurança).

Se o banco de dados estiver distribuído através de vários volumes (por exemplo, os arquivos e dados e o registro do WAL em discos diferentes) pode ser que não haja nenhuma forma de obter instantâneos congelados simultâneos de todos os volumes. A documentação do sistema de arquivos deve ser lida com muita atenção antes de acreditar na técnica de instantâneo consistente em uma situação como esta. A abordagem mais segura é parar o servidor de banco de dados pelo tempo necessário para estabelecer todos os instantâneos congelados.

Deve ser observado que a cópia de segurança do sistema de arquivos não será necessariamente menor que a do Método SQL-dump. Ao contrário, é mais provável que seja maior; por exemplo, o `pg_dump` não necessita fazer cópia de segurança dos índices, mas apenas dos comandos para recriá-los.

### 22.3. Cópia de segurança em-linha

Durante todo o tempo, o PostgreSQL mantém o *registro de escrita prévia* (WAL = *write ahead log*) no subdiretório `pg_xlog` do diretório de dados do agrupamento. O WAL contém todas as alterações realizadas nos arquivos de dados do banco de dados. O WAL existe, principalmente, com a finalidade de fornecer segurança contra quedas: se o sistema cair, o banco de dados pode retornar a um estado consistente “refazendo” as entradas gravadas desde o último ponto de controle. Entretanto, a existência do WAL torna possível uma terceira estratégia para fazer cópia de segurança de banco de dados: pode ser combinada a cópia de segurança do banco de dados no nível de sistema de arquivos, com cópia dos arquivos de segmento do WAL. Se for necessário fazer a recuperação, pode ser feita a recuperação da cópia de segurança do banco de dados no nível de sistema de arquivos e, depois, refeitas as alterações a partir da cópia dos arquivos de segmento do WAL, para trazer a restauração para o tempo presente. A administração desta abordagem é mais complexa que a administração das abordagens anteriores, mas existem alguns benefícios significativos:

- O ponto de partida não precisa ser uma cópia de segurança totalmente consistente. Toda inconsistência interna na cópia de segurança é corrigida quando o WAL é refeito (o que não é muito diferente do que acontece durante a recuperação de uma queda). Portanto, não é necessário um sistema operacional com capacidade de tirar instantâneos, basta apenas o tar, ou outra ferramenta semelhante.
- Como pode ser reunida uma seqüência indefinidamente longa de arquivos de segmento do WAL para serem refeitos, pode ser obtida uma cópia de segurança contínua simplesmente continuando a fazer cópias dos arquivos de segmento do WAL. Isto é particularmente útil para bancos de dados grandes, onde pode não ser conveniente fazer cópias de segurança completas regularmente.
- Não existe nada que diga que as entradas do WAL devem ser refeitas até o fim. Pode-se parar de refazer em qualquer ponto, e obter um instantâneo consistente do banco de dados como se tivesse sido tirado no instante da parada. Portanto, esta técnica suporta a *recuperação para um determinado ponto no tempo*: é possível restaurar voltando o banco de dados para o estado em que se encontrava a qualquer instante posterior ao da realização da cópia de segurança base.
- Se outra máquina, carregada com a mesma cópia de segurança base do banco de dados, for alimentada continuamente com a série de arquivos de segmento do WAL, será criado um sistema reserva à quente (*hot standby*): a qualquer instante esta outra máquina pode ser ativada com uma cópia quase atual do banco de dados.

Da mesma forma que o método de cópia de segurança no nível de sistema de arquivos simples, este método suporta apenas a restauração de todo o agrupamento de bancos de dados, e não a restauração de apenas um subconjunto deste. Requer, também, grande volume de armazenamento de arquivos: a cópia de segurança base pode ser grande, e um sistema carregado gera vários megabytes de tráfego para o WAL que precisam ser guardados. Ainda assim, é o método de cópia de segurança preferido para muitas situações onde é necessária uma alta confiabilidade.

Para fazer uma recuperação bem-sucedida utilizando cópia de segurança em-linha, é necessária uma seqüência contínua de arquivos de segmento do WAL guardados, que venha desde, pelo menos, o instante em que foi feita a cópia de segurança

base do banco de dados. Para começar, deve ser configurado e testado o procedimento para fazer cópia dos arquivos de segmento do WAL, *antes* de ser feita a cópia de segurança base do banco de dados. Assim sendo, primeiro será explicada a mecânica para fazer cópia dos arquivos de segmento do WAL.

### 22.3.1. Cópia dos arquivos de segmento do WAL

Em um sentido abstrato, a execução do sistema PostgreSQL produz uma sequência indefinidamente longa de entradas no WAL. O sistema divide fisicamente esta sequência em *arquivos de segmento* do WAL, normalmente com 16 MB cada (embora o tamanho possa ser alterado durante a construção do PostgreSQL). São atribuídos nomes numéricos aos arquivos de segmento para refletir sua posição na sequência abstrata do WAL. Quando não é feita cópia dos arquivos de segmento do WAL, normalmente o sistema cria apenas uns poucos arquivos de segmento e, depois, “recicla-os” renomeando os arquivos que não são mais de interesse com número de segmento mais alto. Assume-se não existir mais interesse em um arquivo de segmento cujo conteúdo preceda o ponto de controle anterior ao último, podendo, portanto, ser reciclado.

Quando é feita a cópia dos arquivos de segmento do WAL, deseja-se capturar o conteúdo de cada arquivo quando este é completado, guardando os dados em algum lugar antes do arquivo de segmento ser reciclado para ser reutilizado. Dependendo da aplicação e dos periféricos disponíveis, podem haver muitas maneiras de “guardar os dados em algum lugar”: os arquivos de segmento podem ser copiados para outra máquina usando um diretório NFS montado, podem ser escritos em uma unidade de fita (havendo garantia que os arquivos poderão ser restaurados com seus nomes originais), podem ser agrupados e gravados em CD, ou de alguma outra forma. Para que o administrador de banco de dados tenha a máxima flexibilidade possível, o PostgreSQL tenta não assumir nada sobre como as cópias serão feitas. Em vez disso, o PostgreSQL deixa o administrador escolher o comando a ser executado para copiar o arquivo de segmento completado para o local de destino. O comando pode ser tão simples como `cp`, ou pode envolver um script complexo para o interpretador de comandos — tudo depende do administrador.

O comando a ser executado é especificado através do parâmetro de configuração `archive_command`, que na prática é sempre colocado no arquivo `postgresql.conf`. Na cadeia de caracteres do comando, todo `%p` é substituído pelo caminho absoluto do arquivo a ser copiado, enquanto todo `%f` é substituído pelo nome do arquivo apenas. Se for necessário incorporar o caractere `%` ao comando, deve ser escrito `%%`. A forma mais simples de um comando útil é algo como

```
archive_command = 'cp -i %p /mnt/servidor/dir_copias/%f </dev/null'
```

que irá copiar os arquivos de segmento do WAL, prontos para serem copiados, para o diretório `/mnt/servidor/dir_copias` (Isto é um exemplo, e não uma recomendação, e pode não funcionar em todas as plataformas).

O comando para realizar a cópia é executado sob a propriedade do mesmo usuário que está executando o servidor PostgreSQL. Como a série de arquivos do WAL contém efetivamente tudo que está no banco de dados, deve haver certeza que a cópia está protegida contra olhos curiosos; por exemplo, colocando a cópia em diretório sem acesso para grupo ou para todos.

É importante que o comando para realizar a cópia retorne o status de saída zero se, e somente se, for bem-sucedido. Ao receber o resultado zero, o PostgreSQL assume que a cópia do arquivo de segmento do WAL foi bem-sucedida, e remove ou recicla o arquivo de segmento. Entretanto, um status diferente de zero informa ao PostgreSQL que o arquivo não foi copiado; serão feitas tentativas periódicas até ser bem-sucedida.

Geralmente o comando de cópia deve ser projetado de tal forma que não sobrescreva algum arquivo de cópia pré-existente. Esta é uma característica de segurança importante para preservar a integridade da cópia no caso de um erro do administrador (tal como enviar a saída de dois servidores diferentes para o mesmo diretório de cópias). Aconselha-se a testar o comando de cópia proposto para ter certeza que não sobrescreve um arquivo existente, *e que retorna um status diferente de zero neste caso*. Tem sido observado que `cp -i` faz isto corretamente em algumas plataformas, mas não em outras. Se o comando escolhido não tratar este caso corretamente por conta própria, deve ser adicionado um comando para testar a existência do arquivo de cópia. Por exemplo, algo como

```
archive_command = 'test ! -f ../%f && cp %p ../%f'
```

funciona corretamente na maioria das variantes do Unix.

Ao projetar a configuração de cópia deve ser considerado o que vai acontecer quando o comando de cópia falhar repetidas vezes, seja porque alguma funcionalidade requer intervenção do operador, ou porque não há espaço para armazenar a cópia. Esta situação pode ocorrer, por exemplo, quando a cópia é escrita em fita e não há um sistema automático para troca de

fitas: quando a fita ficar cheia, não será possível fazer outras cópias enquanto a fita não for trocada. Deve-se garantir que qualquer condição de erro, ou solicitação feita a um operador humano, seja relatada de forma apropriada para que a situação possa ser resolvida o mais rápido possível. Enquanto a situação não for resolvida, continuarão sendo criados novos arquivos de segmento do WAL no diretório `pg_xlog`.

A velocidade do comando de cópia não é importante, desde que possa acompanhar a taxa média de geração de dados para o WAL. A operação normal prossegue mesmo que o processo de cópia fique um pouco atrasado. Se o processo de cópia ficar muito atrasado, vai aumentar a quantidade de dados perdidos caso ocorra um desastre. Significa, também, que o diretório `pg_xlog` vai conter um número grande de arquivos de segmento que ainda não foram copiados, podendo, inclusive, exceder o espaço livre em disco. Aconselha-se que o processo de cópia seja monitorado para garantir que esteja funcionando da forma planejada.

Havendo preocupação em se poder recuperar até o presente instante, devem ser efetuados passos adicionais para garantir que o arquivo de segmento do WAL corrente, parcialmente preenchido, também seja copiado para algum lugar. Isto é particularmente importante no caso do servidor gerar pouco tráfego para o WAL (ou tiver períodos ociosos onde isto acontece), uma vez que pode levar muito tempo até que o arquivo de segmento fique totalmente preenchido e pronto para ser copiado. Uma forma possível de tratar esta situação é definir uma entrada no cron<sup>1</sup> que periodicamente, talvez uma vez por minuto, identifique o arquivo de segmento do WAL corrente e o guarde em algum lugar seguro. Então, a combinação dos arquivos de segmento do WAL guardados, com o arquivo de segmento do WAL corrente guardado, será suficiente para garantir que o banco de dados pode ser restaurado até um minuto, ou menos, antes do presente instante. Atualmente este comportamento não está presente no PostgreSQL, porque não se deseja complicar a definição de `archive_command` requerendo que este acompanhe cópias bem-sucedidas, mas diferentes, do mesmo arquivo do WAL. O `archive_command` é chamado apenas para segmentos do WAL completados. Exceto no caso de novas tentativas devido a falha, só é chamado uma vez para um determinado nome de arquivo.

Ao escrever o comando de cópia, deve ser assumido que os nomes dos arquivos a serem copiados podem ter comprimento de até 64 caracteres, e que podem conter qualquer combinação de letras ASCII, dígitos e pontos. Não é necessário recordar o caminho original completo (%p), mas é necessário recordar o nome do arquivo (%f).

Deve ser lembrado que embora a cópia do WAL permita restaurar toda modificação feita nos dados dos bancos de dados do PostgreSQL, não restaura as alterações feitas nos arquivos de configuração (ou seja, nos arquivos `postgresql.conf`, `pg_hba.conf` e `pg_ident.conf`), uma vez que estes arquivos são editados manualmente, em vez de através de operações SQL. Aconselha-se a manter os arquivos de configuração em um local onde são feitas cópias de segurança regulares do sistema de arquivos. Para mudar os arquivos de configuração de lugar, deve ser consultada a Seção 16.4.1.

### 22.3.2. Criação da cópia de segurança base

O procedimento para fazer a cópia de segurança base é relativamente simples:

1. Garantir que a cópia dos arquivos de segmento do WAL esteja habilitada e funcionando.
2. Conectar ao banco de dados como um superusuário e executar o comando

```
SELECT pg_start_backup('rótulo');
```

onde `rótulo` é qualquer cadeia de caracteres que se deseje usar para identificar unicamente esta operação de cópia de segurança (Uma boa prática é utilizar o caminho completo de onde se deseja colocar o arquivo de cópia de segurança). A função `pg_start_backup` cria o arquivo *rótulo da cópia de segurança*, chamado `backup_label`, com informações sobre a cópia de segurança, no diretório do agrupamento.

Para executar este comando, não importa qual banco de dados do agrupamento é usado para fazer a conexão. O resultado retornado pela função pode ser ignorado; mas se for relatado um erro, este deve ser tratado antes de prosseguir.

3. Realizar a cópia de segurança utilizando qualquer ferramenta conveniente para cópia de segurança do sistema de arquivos, como `tar` ou `cpio`. Não é necessário, nem desejado, parar a operação normal do banco de dados enquanto a cópia é feita.
4. Conectar novamente ao banco de dados como um superusuário e executar o comando:

```
SELECT pg_stop_backup();
```

Se a execução for bem sucedida, está terminado.



Não é necessário ficar muito preocupado com o tempo decorrido entre a execução de `pg_start_backup` e o início da realização da cópia de segurança, nem entre o fim da realização da cópia de segurança e a execução de `pg_stop_backup`; uns poucos minutos de atraso não vão criar nenhum problema. Entretanto, deve haver certeza que as operações são realizadas sequencialmente, sem que haja sobreposição.

Deve haver certeza que a cópia de segurança inclui todos os arquivos sob o diretório do agrupamento de bancos de dados (por exemplo, `/usr/local/pgsql/data`). Se estiverem sendo utilizados espaços de tabelas que não residem sob este diretório, deve-se ter o cuidado de incluí-los também (e ter certeza que a cópia de segurança guarda vínculos simbólicos como vínculos, senão a restauração vai danificar os espaços de tabelas).

Entretanto, podem ser omitidos da cópia de segurança os arquivos sob o subdiretório `pg_xlog` do diretório do agrupamento. Esta pequena complicação a mais vale a pena ser feita porque reduz o risco de erros na restauração. É fácil de ser feito se `pg_xlog` for um vínculo simbólico apontando para algum lugar fora do agrupamento, o que é uma configuração comum por razões de desempenho.

Para poder utilizar esta cópia de segurança base, devem ser mantidas por perto todas as cópias dos arquivos de segmento do WAL gerados no momento ou após o início da mesma. Para ajudar a realizar esta tarefa, a função `pg_stop_backup` cria o *arquivo de história de cópia de segurança*, que é armazenado imediatamente na área de cópia do WAL. Este arquivo recebe um nome derivado do primeiro arquivo de segmento do WAL que é necessário possuir para fazer uso da cópia de segurança. Por exemplo, se o arquivo do WAL tiver o nome `0000000100001234000055CD`, o arquivo de história de cópia de segurança vai ter um nome parecido com `0000000100001234000055CD.007C9330.backup` (A segunda parte do nome do arquivo representa a posição exata dentro do arquivo do WAL, podendo normalmente ser ignorada). Uma vez que o arquivo contendo a cópia de segurança base tenha sido guardado em local seguro, podem ser apagados todos os arquivos de segmento do WAL com nomes numericamente precedentes a este número. O arquivo de história de cópia de segurança é apenas um pequeno arquivo texto. Contém a cadeia de caracteres rótulo fornecida à função `pg_start_backup`, assim como as horas de início e fim da cópia de segurança. Se o rótulo for utilizado para identificar onde está armazenada a cópia de segurança base do banco de dados, então basta o arquivo de história de cópia de segurança para se saber qual é o arquivo de cópia de segurança a ser restaurado, no caso disto precisar ser feito.

Uma vez que é necessário manter por perto todos os arquivos de segmento do WAL copiados desde a última cópia de segurança base, o intervalo entre estas cópias de segurança geralmente deve ser escolhido tendo por base quanto armazenamento se deseja consumir para os arquivos do WAL guardados. Também deve ser considerado quanto tempo se está preparado para despendar com a restauração, no caso de ser necessário fazer uma restauração — o sistema terá que refazer todos os segmentos do WAL, o que pode ser muito demorado se tiver sido decorrido muito tempo desde a última cópia de segurança base.

Também vale a pena notar que a função `pg_start_backup` cria no diretório do agrupamento de bancos de dados um arquivo chamado `backup_label`, que depois é removido pela função `pg_stop_backup`. Este arquivo fica guardado como parte do arquivo de cópia de segurança base. O arquivo rótulo de cópia de segurança inclui a cadeia de caracteres rótulo fornecida para a função `pg_start_backup`, assim como a hora em que `pg_start_backup` foi executada, e o nome do arquivo de segmento inicial do WAL. Em caso de dúvida, é possível olhar dentro do arquivo de cópia de segurança base e determinar com exatidão de qual sessão de cópia de segurança este arquivo provém.

Também é possível fazer a cópia de segurança base enquanto o postmaster está parado. Neste caso, obviamente não podem ser utilizadas as funções `pg_start_backup` e `pg_stop_backup`, sendo responsabilidade do administrador controlar a que cópia de segurança cada arquivo pertence, e até quanto tempo atrás os arquivos de segmento do WAL associados vão. Geralmente é melhor seguir os procedimentos para cópia de segurança mostrados acima.

### 22.3.3. Recuperação a partir de cópia de segurança em-linha

Certo, aconteceu o pior e é necessário recuperar a partir da cópia de segurança. O procedimento está mostrado abaixo:

1. Parar o postmaster, se estiver executando.
2. Havendo espaço para isso, copiar todo o diretório de dados do agrupamento, e todos os espaços de tabelas, para um lugar temporário, para o caso de necessidade. Deve ser observado que esta medida de precaução requer a existência de espaço no sistema suficiente para manter duas cópias do banco de dados existente. Se não houver espaço suficiente, é necessário pelo menos uma cópia do conteúdo do subdiretório `pg_xlog` do diretório de dados do agrupamento, porque pode conter arquivos de segmento do WAL que não foram copiados quando o sistema parou.
3. Apagar todos os arquivos e subdiretórios existentes sob o diretório de dados do agrupamento, e sob os diretórios raiz dos espaços de tabelas em uso.

4. Restaurar os arquivos do banco de dados a partir da cópia de segurança base. Deve-se tomar cuidado para que sejam restaurados com o dono correto (o usuário do sistema de banco de dados, e não o usuário `root`), e com as permissões corretas. Se estiverem sendo utilizados espaços de tabelas, deve ser verificado se foram restaurados corretamente os vínculos simbólicos no subdiretório `pg_tblspc/`.
5. Remover todos os arquivos presentes no subdiretório `pg_xlog`; porque estes vêm da cópia de segurança base e, portanto, provavelmente estão obsoletos. Se o subdiretório `pg_xlog` não fizer parte da cópia de segurança base, então este subdiretório deve ser criado, assim como o subdiretório `pg_xlog/archive_status`.
6. Se existirem arquivos de segmento do WAL que não foram copiados para o diretório de cópias, mas que foram salvos no passo 2, estes devem ser copiados para o diretório `pg_xlog`; é melhor copiá-los em vez de movê-los, para que ainda existam arquivos não modificados caso ocorra algum problema e o processo tenha de ser recomeçado.
7. Criar o arquivo de comando de recuperação `recovery.conf` no diretório de dados do agrupamento (consulte Recovery Settings). Também pode ser útil modificar temporariamente o arquivo `pg_hba.conf`, para impedir que os usuários comuns se conectem até que se tenha certeza que a recuperação foi bem-sucedida.
8. Iniciar o postmaster. O postmaster vai entrar no modo de recuperação e prosseguir lendo os arquivos do WAL necessários. Após o término do processo de recuperação, o postmaster muda o nome do arquivo `recovery.conf` para `recovery.done` (para impedir que entre novamente no modo de recuperação no caso de uma queda posterior), e depois começa as operações normais de banco de dados.
9. Deve ser feita a inspeção do conteúdo do banco de dados para garantir que a recuperação foi feita até onde deveria ser feita. Caso contrário, deve-se retornar ao passo 1. Se tudo correu bem, liberar o acesso aos usuários retornando `pg_hba.conf` à sua condição normal.

A parte chave de todo este procedimento é a definição do arquivo contendo o comando de recuperação, que descreve como se deseja fazer a recuperação, e até onde a recuperação deve ir. Pode ser utilizado o arquivo `recovery.conf.sample` (geralmente presente no diretório de instalação `share`) na forma de um protótipo <sup>2</sup>. O único parâmetro requerido no arquivo `recovery.conf` é `restore_command`, que informa ao PostgreSQL como trazer de volta os arquivos de segmento do WAL copiados. Como no `archive_command`, este parâmetro é uma cadeia de caracteres para o interpretador de comandos. Pode conter `%f`, que é substituído pelo nome do arquivo do WAL a ser trazido de volta, e `%p`, que é substituído pelo caminho absoluto para onde o arquivo do WAL será copiado. Se for necessário incorporar o caractere `%` ao comando, deve ser escrito `%%`. A forma mais simples de um comando útil é algo como

```
restore_command = 'cp /mnt/servidor/dir_copias/%f %p'
```

que irá copiar os arquivos de segmento do WAL previamente guardados a partir do diretório `/mnt/servidor/dir_copias`. É claro que pode ser utilizado algo muito mais complicado, talvez um script que solicite ao operador a montagem da fita apropriada.

É importante que o comando retorne um status de saída diferente de zero em caso de falha. Será solicitado ao comando os arquivos do WAL cujos nomes não estejam presente entre as cópias; deve retornar um status diferente de zero quando for feita a solicitação. Esta não é uma condição de erro. Deve-se tomar cuidado para que o nome base do caminho `%p` seja diferente de `%f`; não deve ser esperado que sejam intercambiáveis.

Os arquivos de segmento do WAL que não puderem ser encontrados entre as cópias, serão procurados no diretório `pg_xlog/`; isto permite que os arquivos de segmento recentes, ainda não copiados, sejam utilizados. Entretanto, os arquivos de segmento que estiverem entre as cópias terão preferência sobre os arquivos em `pg_xlog`. O sistema não sobrescreve os arquivos presentes em `pg_xlog` quando busca os arquivos guardados.

Normalmente a recuperação prossegue através de todos os arquivos de segmento do WAL, portanto restaurando o banco de dados até o presente momento (ou tão próximo quanto se pode chegar utilizando os segmentos do WAL). Mas se for necessário recuperar até algum ponto anterior no tempo (digamos, logo antes do DBA júnior ter apagado a tabela principal de transação de alguém), deve-se simplesmente especificar no arquivo `recovery.conf` o ponto de parada requerido. O ponto de parada, conhecido como “destino da recuperação”, pode ser especificado tanto pela data e hora quanto pelo término de um ID de transação específico. Até o momento em que este manual foi escrito, somente podia ser utilizada a opção data e hora, pela falta de ferramenta para ajudar a descobrir com precisão o identificador de transação a ser utilizado.

**Nota:** O ponto de parada deve estar situado após o momento de término da cópia de segurança base (o momento em que foi executada a função `pg_stop_backup`). A cópia de segurança não pode ser utilizada para recuperar até um

momento em que a cópia de segurança base estava em andamento (Para recuperar até este ponto, deve-se retornar para uma cópia de segurança base anterior e refazer a partir desta cópia).

### 22.3.3.1. Definições de recuperação

Estas definições somente podem ser feitas no arquivo `recovery.conf`, e são aplicadas apenas pela duração da recuperação. Devem ser definidas novamente nas próximas recuperações que se desejar realizar. As definições não podem ser alteradas após a recuperação ter começado.

`restore_command(string)`

O comando, para o interpretador de comandos, a ser executado para trazer de volta os segmentos da série de arquivos do WAL guardados. Este parâmetro é requerido. Todo `%f` presente na cadeia de caracteres é substituído pelo nome do arquivo a ser trazido de volta das cópias, e todo `%p` é substituído pelo caminho absoluto para onde o arquivo será copiado no servidor. Se for necessário incorporar o caractere `%` ao comando, deve ser escrito `%%`.

É importante que o comando retorne o status de saída zero se, e somente se, for bem-sucedido. Será solicitado ao comando os arquivos cujos nomes não estejam presentes entre as cópias; deve retornar um status diferente de zero quando for feita a solicitação. Exemplos:

```
restore_command = 'cp /mnt/servidor/dir_copias/%f "%p"'
restore_command = 'copy /mnt/servidor/dir_copias/%f "%p"' # Windows
```

`recovery_target_time(timestamp)`

Este parâmetro especifica o carimbo do tempo até onde a recuperação deve prosseguir. Somente pode ser especificado um entre `recovery_target_time` e `recovery_target_xid`. O padrão é recuperar até o fim do WAL. O ponto de parada preciso também é influenciado por `recovery_target_inclusive`.

`recovery_target_xid(string)`

Este parâmetro especifica o identificador de transação até onde a recuperação deve prosseguir. Deve-se ter em mente que enquanto os identificadores são atribuídos sequencialmente no início da transação, as transações podem ficar completas em uma ordem numérica diferente. As transações que serão recuperadas são aquelas que foram efetivadas antes (e opcionalmente incluindo) a transação especificada. Somente pode ser especificado um entre `recovery_target_xid` e `recovery_target_time`. O padrão é recuperar até o fim do WAL. O ponto de parada preciso também é influenciado por `recovery_target_inclusive`.

`recovery_target_inclusive(boolean)`

Especifica se a parada deve acontecer logo após o destino de recuperação especificado (`true`), ou logo antes do destino de recuperação especificado (`false`). Aplica-se tanto a `recovery_target_time` quanto a `recovery_target_xid`, o que for especificado para esta recuperação. Indica se as transações que possuem exatamente a hora de efetivação ou o identificador de destino, respectivamente, serão incluídas na recuperação. O valor padrão é `true`.

`recovery_target_timeline(string)`

Especifica a recuperação de uma determinada cronologia. O padrão é recuperar ao longo da cronologia que era a cronologia corrente quando foi feita a cópia de segurança base. Somente será necessário definir este parâmetro em situações de re-recuperações complexas, onde é necessário retornar para um estado a que se chegou após uma recuperação para um ponto no tempo. Consulte a explicação na Seção 22.3.4.

### 22.3.4. Cronologias

A capacidade de restaurar um banco de dados para um determinado ponto anterior no tempo cria algumas complexidades que são semelhantes às das histórias de ficção científica sobre viagem no tempo e universos paralelos. Por exemplo, na história original do banco de dados talvez tenha sido removida uma tabela importante às 5:15 da tarde de terça-feira. Imperturbável, o administrador pega a cópia de segurança e faz uma restauração para o ponto no tempo 5:14 da tarde de terça-feira, e o sistema volta a funcionar. *Nesta* história do universo do banco de dados, a tabela nunca foi removida. Mas suponha que mais tarde seja descoberto que esta não foi uma boa idéia, e que se deseja voltar para algum ponto posterior da história original. Mas isto não poderá ser feito, porque quando o banco de dados foi posto em atividade este sobrescreveu alguns dos arquivos de segmento do WAL que levariam ao ponto no tempo onde agora quer se chegar. Portanto, realmente é necessário fazer distinção entre a série de entradas no WAL geradas após a recuperação para um ponto no tempo, e aquelas geradas durante a história original do banco de dados.

Para lidar com estes problemas, o PostgreSQL possui a noção de *cronologias* (*timelines*). Cada vez que é feita uma recuperação no tempo anterior ao fim da sequência do WAL, é criada uma nova cronologia para identificar a série de registros do WAL geradas após a recuperação (entretanto, se a recuperação prosseguir até o final do WAL, não é criada uma nova cronologia: apenas se estende a cronologia existente). O número identificador da cronologia é parte dos nomes dos arquivos de segmento do WAL e, portanto, uma nova cronologia não sobrescreve os dados do WAL gerados pelas cronologias anteriores. É possível, na verdade, guardar muitas cronologias diferentes. Embora possa parecer uma funcionalidade sem utilidade, muitas vezes é de grande valia. Considere a situação onde não se tem certeza absoluta de até que ponto no tempo deve ser feita a recuperação e, portanto, devem ser feitas muitas tentativas de recuperação até ser encontrado o melhor lugar para se desviar da história antiga. Sem as cronologias este processo em pouco tempo cria uma confusão impossível de ser gerenciada. Com as cronologias, pode ser feita a recuperação até *qualquer* estado anterior, inclusive os estados no desvio de cronologia abandonados posteriormente.

Toda vez que é criada uma nova cronologia, o PostgreSQL cria um arquivo de “história da cronologia” que mostra de que cronologia foi feito o desvio, e quando. Os arquivos de cronologia são necessários para permitir o sistema buscar os arquivos de segmento do WAL corretos ao fazer a recuperação a partir de uma área de cópias que contém várias cronologias. Portanto, estes arquivos são guardados na área de cópias como qualquer arquivo de segmento do WAL. Os arquivos de cronologia são apenas pequenos arquivos texto sendo, portanto, barato e apropriado mantê-los guardados indefinidamente (ao contrário dos arquivos de segmento que são grandes). É possível, caso se deseje fazê-lo, adicionar comentários aos arquivos de cronologia para fazer anotações personalizadas sobre como e porque foi criada uma determinada cronologia. Estes comentários são muito úteis quando há um grande número de cronologias criadas como resultado de experiências.

O comportamento padrão de recuperação é recuperar ao longo da cronologia que era a cronologia corrente quando foi feita a cópia de segurança base. Se for desejado fazer a recuperação utilizando uma cronologia filha (ou seja, deseja-se retornar para algum estado que foi gerado após uma tentativa de recuperação), é necessário especificar o identificador da cronologia de destino no arquivo `recovery.conf`. Não é possível fazer a recuperação para um estado que foi um desvio anterior à cópia de segurança base.

### 22.3.5. Cuidado

No momento em que esta documentação foi escrita, haviam várias limitações para o método de cópia de segurança em-linha, que provavelmente serão corrigidas nas versões futuras:

- Atualmente não são gravadas no WAL as operações em índices que não são B-tree (índices hash, R-tree e GiST), portanto quando o WAL é refeito os índices destes tipos não são atualizados. A prática recomendada para contornar este problema é reindexar manualmente todos os índices destes tipos após o término da operação de recuperação.

Também deve ser notado que o formato atual do WAL é muito volumoso, uma vez que inclui muitos instantâneos de páginas de disco. Isto é apropriado para a finalidade de recuperação de quedas, uma vez que pode ser necessário corrigir páginas de disco parcialmente preenchidas. Entretanto, não é necessário armazenar tantas cópias de páginas para operações de recuperação para um determinado ponto no tempo. Uma área para desenvolvimento futuro é a compressão dos dados do WAL copiados, pela remoção das cópias de página desnecessárias.

## 22.4. Migração entre versões

Esta seção explica como migrar o banco de dados de uma versão do PostgreSQL para outra versão mais nova. O procedimento de instalação do PostgreSQL não é assunto desta seção; estes detalhes se encontram no Capítulo 14.

Como regra geral, o formato interno de armazenamento dos dados está sujeito a alterações entre versões principais do PostgreSQL (onde muda o número após o primeiro ponto). Isto não se aplica às versões secundárias sob a mesma versão principal (onde muda o número após o segundo ponto); estas versões sempre possuem formatos de armazenamento compatíveis. Por exemplo, as versões 7.0.1, 7.1.2 e 7.2 não são compatíveis, enquanto as versões 7.1.1 e 7.1.2 são compatíveis. Quando é feita a atualização para uma versão compatível, pode-se simplesmente substituir os executáveis e reutilizar o diretório de dados no disco. Caso contrário, é necessário fazer a cópia de segurança dos dados e “restaurá-la” no novo servidor. A cópia de segurança tem de ser feita utilizando o `pg_dump`; como é óbvio, os métodos de cópia de segurança no nível de sistema operacional não funcionam. São realizadas verificações que impedem que seja utilizado um diretório de dados de uma versão incompatível do PostgreSQL e, portanto, não pode ser causado nenhum grande dano quando se tenta ativar uma versão errada do servidor em um diretório de dados.

É recomendada a utilização dos programas `pg_dump` e `pg_dumpall` da nova versão do PostgreSQL, para tirar vantagem das melhorias que podem ter sido introduzidas nestes programas. As versões corrente dos programas de cópia de segurança podem ler os dados de qualquer versão do servidor da 7.0 em diante.

O menor tempo de parada pode ser obtido instalando o novo servidor em um diretório diferente, e executando tanto o servidor novo quanto o antigo em paralelo, em portas diferentes. Depois pode ser executado algo como

```
pg_dumpall -p 5432 | psql -d template1 -p 6543
```

para transferir os dados. Se for desejado, pode ser utilizado um arquivo intermediário. Depois o servidor antigo pode ser parado, e o novo servidor ativado na mesma porta que o servidor antigo estava utilizando. Deve haver certeza que o banco de dados antigo não foi atualizado após ser executado o `pg_dumpall`, senão obviamente serão perdidos dados. Para obter informações sobre como proibir o acesso deve ser visto o Capítulo 19.

Na prática, provavelmente será desejado testar os aplicativos cliente na nova configuração antes de trocar de versão. Este é outro motivo para configurar instalações simultâneas para as versões antiga e nova.

Se não for possível ou não for desejado executar dois servidores em paralelo, pode ser realizada a etapa de cópia de segurança antes de instalar a nova versão, parar o servidor, mover a versão antiga para outro lugar, instalar a nova versão, ativar o novo servidor, e restaurar os dados. Por exemplo:

```
pg_dumpall > backup
pg_ctl stop
mv /usr/local/pgsql /usr/local/pgsql.old
cd ~/postgresql-8.0.0
gmake install
initdb -D /usr/local/pgsql/data
postmaster -D /usr/local/pgsql/data
psql template1 < backup
```

As formas de parar e ativar o servidor, e outros detalhes, devem ser vistos no Capítulo 16. As instruções de instalação dão conselhos sobre os lugares estratégicos para realizar estes passos.

**Nota:** Quando “a instalação antiga é movida para outro lugar”, talvez não seja mais totalmente utilizável. Alguns programas executáveis contêm caminhos absolutos para vários programas instalados e arquivos de dados. Isto normalmente não é um grande problema, mas se for planejado utilizar duas instalações em paralelo por um período de tempo, devem ser atribuídos diretórios de instalação diferentes em tempo de construção (Este problema foi corrigido na versão 8.0 e posteriores do PostgreSQL, mas deve-se estar atento ao mesmo quando instalações mais antigas são movidas de lugar).

## Notas

1. `cron` — processo (daemon) para executar comandos agendados. (N. do T.)
2. O arquivo `recovery.conf.sample` está presente no diretório `/src/backend/access/transam` da distribuição do código fonte e do CVS. (N. do T.)

## Capítulo 23. Monitoramento das atividades do banco de dados

Freqüentemente o administrador de banco de dados deseja saber, “O que o sistema está fazendo agora?”. Este capítulo explica como descobrir isto.

Estão disponíveis várias ferramentas para monitorar a atividade do banco de dados e analisar o desempenho. A maior parte deste capítulo dedica-se a descrever o coletor de estatísticas do PostgreSQL, mas não se deve desprezar os programas regulares de monitoramento do Unix, como `ps`, `top`, `iostat` e `vmstat`. Também, uma vez que tenha sido identificado um comando com baixo desempenho, podem ser necessárias outras investigações utilizando o comando `EXPLAIN` do PostgreSQL. A Seção 13.1 discute o comando `EXPLAIN` e outros métodos para compreender o comportamento individual de um comando.

### 23.1. Ferramentas padrão do Unix

O PostgreSQL modifica, na maioria das plataformas, o seu título de comando mostrado pelo `ps` de forma que os processos servidor individuais possam ser prontamente identificados. Um exemplo do que é mostrado é:

```
$ ps auxww | grep ^postgres
```

```
postgres  960  0.0  1.1  6104 1480 pts/1    SN   13:17   0:00 postmaster -i
postgres  963  0.0  1.1  7084 1472 pts/1    SN   13:17   0:00 postgres: stats buffer process
postgres  965  0.0  1.1  6152 1512 pts/1    SN   13:17   0:00 postgres: stats collector process
postgres  998  0.0  2.3  6532 2992 pts/1    SN   13:18   0:00 postgres: tgl runbug 127.0.0.1 idle
postgres 1003  0.0  2.4  6532 3128 pts/1    SN   13:19   0:00 postgres: tgl regression [local] SELECT
waiting
postgres 1016  0.1  2.4  6532 3080 pts/1    SN   13:19   0:00 postgres: tgl regression [local] idle in
transaction
```

A forma apropriada para chamar o `ps` varia entre plataformas diferentes, assim como os detalhes mostrados. Este exemplo foi tirado de um sistema Linux recente. O primeiro processo listado neste exemplo é o `postmaster`, o processo servidor mestre. Os argumentos do comando mostrados são os mesmos fornecidos quando o `postmaster` foi ativado. Os dois processos seguintes implementam o coletor de estatísticas, que será descrito em detalhes na próxima seção (Não estão presentes quando o sistema é configurado para não ativar o coletor de estatísticas). Cada um dos demais processos é um processo servidor tratando uma conexão cliente. Cada um destes processos define a exibição da sua linha de comando na forma:

```
postgres: usuário banco_de_dados hospedeiro atividade
```

Os itens usuário, banco de dados e hospedeiro originais da conexão permanecem o mesmo durante toda a existência da conexão cliente, mas o indicador de atividade muda.<sup>1 2</sup> A atividade pode ser `idle` (ou seja, ociosa, aguardando por um comando do cliente), `idle in transaction` (aguardando pelo cliente dentro de um bloco `BEGIN`), ou o nome de um tipo de comando como `SELECT`. Além disso, é anexado `waiting` se o processo servidor estiver aguardando no momento por um bloqueio mantido por outro processo servidor. No exemplo acima pode ser inferido que o processo 1003 está aguardando o processo 1016 completar sua transação e, portanto, liberar algum bloqueio.

**Dica:** O Solaris requer tratamento especial. Deve ser utilizado `/usr/ucb/ps` em vez de `/bin/ps`. Também devem ser utilizados dois sinalizadores `w`, e não apenas um. Além disso, a chamada original do comando `postmaster` deve possuir um status exibido pelo `ps` mais curto que o exibido por cada processo servidor. Se estas três coisas não forem feitas, a saída do `ps` para cada processo servidor será a linha de comando original do `postmaster`.

### 23.2. O coletor de estatísticas

O *coletor de estatísticas* do PostgreSQL é um subsistema de apoio a coleta e relatório de informações sobre as atividades do servidor. Atualmente, o coletor pode contar acessos a tabelas e índices em termos de blocos de disco e linhas individuais. Também apóia a determinação exata do comando sendo executado no momento pelos outros processos servidor.

### 23.2.1. Configuração da coleta de estatísticas

Uma vez que a coleta de estatísticas adiciona alguma sobrecarga à execução do comando, o sistema pode ser configurado para coletar informações, ou não. Isto é controlado por parâmetros de configuração, normalmente definidos no arquivo `postgresql.conf` (para obter detalhes sobre como definir os parâmetros de configuração deve ser consultada a Seção 16.4).

O parâmetro `stats_start_collector` deve ser definido como `true` para que o coletor de estatísticas seja ativado. Esta é a definição padrão e recomendada, mas pode ser desabilitado se não houver interesse nas estatísticas e for desejado eliminar até a última gota de sobrecarga (Entretanto, provavelmente o ganho será pequeno). Deve ser observado que esta opção não pode ser mudada enquanto o servidor está executando.

Os parâmetros `stats_command_string`, `stats_block_level` e `stats_row_level` controlam a quantidade de informação que é enviada para o coletor e, portanto, determinam quanta sobrecarga ocorre em tempo de execução. Determinam se o processo servidor envia para o coletor a cadeia de caracteres do comando corrente, as estatísticas de acesso no nível de bloco de disco, e as estatísticas de acesso no nível de linha, respectivamente. Normalmente estes parâmetros são definidos no arquivo `postgresql.conf` e, portanto, se aplicam a todos os processos servidor, mas é possível ativá-los ou desativá-los para sessões individuais utilizando o comando `SET` (Para evitar que os usuários comuns escondam suas atividades do administrador, somente os superusuários podem alterar estes parâmetros através do comando `SET`).

**Nota:** Uma vez que o valor dos parâmetros `stats_command_string`, `stats_block_level` e `stats_row_level` são `false` por padrão, muito poucas estatísticas são coletadas na configuração padrão. Habilitar uma ou mais destas variáveis de configuração aumenta, significativamente, a quantidade de dados úteis produzidos pelo coletor de estatísticas, ao custo de uma sobrecarga adicional em tempo de execução.

### 23.2.2. Ver as estatísticas coletadas

Estão disponíveis diversas visões pré-definidas para mostrar os resultados das estatísticas coletadas, conforme listado na Tabela 23-1. Como alternativa, podem ser construídas visões personalizadas utilizando as funções de estatísticas subjacentes.

Ao se utilizar as estatísticas para monitorar a atividade corrente, é importante ter em mente que as informações não são atualizadas instantaneamente. Cada processo servidor individual transmite os novos contadores de acesso a bloco e a linha para o coletor logo antes de ficar ocioso; portanto, um comando ou transação ainda em progresso não afeta os totais exibidos. Também, o próprio coletor emite um novo relatório no máximo uma vez a cada `pgstat_stat_interval` milissegundos (500 por padrão). Portanto, as informações mostradas são anteriores à atividade corrente. A informação do comando corrente é enviada para o coletor imediatamente, mas ainda está sujeita ao retardo de `pgstat_stat_interval` antes de se tornar visível.

Outro ponto importante é que, quando se solicita a um processo servidor para mostrar uma destas estatísticas, primeiro este busca os relatórios mais recentes emitidos pelo processo coletor, e depois continua utilizando este instantâneo para todas as visões e funções de estatística até o término da transação corrente. Portanto, as estatísticas parecem não mudar enquanto se permanece na transação corrente. Isto é uma característica, e não um erro, porque permite realizar várias consultas às estatísticas e correlacionar os resultados sem se preocupar com números variando por baixo. Se desejar ver novos resultados a cada consulta, certifique-se que as consultas estão fora de qualquer bloco de transação.

**Tabela 23-1. Visões de estatísticas padrão**

Nome da visão	Descrição
<code>pg_stat_activity</code>	Uma linha por processo servidor, mostrando o ID do processo, o banco de dados, o usuário, o comando corrente e a hora em que o comando corrente começou a executar. As colunas que mostram os dados do comando corrente somente estão disponíveis quando o parâmetro <code>stats_command_string</code> está habilitado. Além disso, estas colunas mostram o valor nulo a menos que o usuário consultando a visão seja um superusuário, ou o mesmo usuário dono do processo sendo mostrado (Deve ser observado que devido ao retardo do que é informado pelo coletor, o comando corrente somente será mostrado no caso dos comandos com longo tempo de execução).
<code>pg_stat_database</code>	Uma linha por banco de dados, mostrando o número de processos servidor ativos,

Nome da visão	Descrição
	total de transações efetivadas e total de transações canceladas neste banco de dados, total de blocos de disco lidos e total de acertos no <code>buffer</code> (ou seja, solicitações de leitura de bloco evitadas por encontrar o bloco no <code>cache</code> do <code>buffer</code> ).
<code>pg_stat_all_tables</code>	Para cada tabela do banco de dados corrente, o número total de: varreduras seqüenciais e de índice; linhas retornadas por cada tipo de varredura; linhas inseridas, atualizadas e excluídas.
<code>pg_stat_sys_tables</code>	O mesmo que <code>pg_stat_all_tables</code> , exceto que somente são mostradas as tabelas do sistema.
<code>pg_stat_user_tables</code>	O mesmo que <code>pg_stat_all_tables</code> , exceto que somente são mostradas as tabelas de usuário.
<code>pg_stat_all_indexes</code>	Para cada índice do banco de dados corrente, o total de varreduras de índice que utilizaram este índice, o número de linhas do índice lidas, e o número de linhas da tabela buscadas com sucesso (Pode ser menor quando existem entradas do índice apontando para linhas da tabela expiradas)
<code>pg_stat_sys_indexes</code>	O mesmo que <code>pg_stat_all_indexes</code> , exceto que somente são mostrados os índices das tabelas do sistema.
<code>pg_stat_user_indexes</code>	O mesmo que <code>pg_stat_all_indexes</code> , exceto que somente são mostrados os índices das tabelas de usuário.
<code>pg_statio_all_tables</code>	Para cada tabela do banco de dados corrente, o número total de blocos de disco da tabela lidos, o número de acertos no <code>buffer</code> , o número de blocos de disco lidos e acertos no <code>buffer</code> para todos os índices da tabela, o número de blocos de disco lidos e acertos no <code>buffer</code> para a tabela auxiliar TOAST da tabela (se houver), e o número de blocos de disco lidos e acertos no <code>buffer</code> para o índice da tabela TOAST.
<code>pg_statio_sys_tables</code>	O mesmo que <code>pg_statio_all_tables</code> , exceto que somente são mostradas as tabelas do sistema.
<code>pg_statio_user_tables</code>	O mesmo que <code>pg_statio_all_tables</code> , exceto que somente são mostradas as tabelas de usuário.
<code>pg_statio_all_indexes</code>	Para cada índice do banco de dados corrente, o número de blocos de disco lidos e de acertos no <code>buffer</code> para o índice.
<code>pg_statio_sys_indexes</code>	O mesmo que <code>pg_statio_all_indexes</code> , exceto que somente são mostrados os índices das tabelas do sistema.
<code>pg_statio_user_indexes</code>	O mesmo que <code>pg_statio_all_indexes</code> , exceto que somente são mostrados os índices das tabelas de usuário.
<code>pg_statio_all_sequences</code>	Para cada objeto de seqüência do banco de dados corrente, o número de blocos de disco lidos e de acertos no <code>buffer</code> para a seqüência.
<code>pg_statio_sys_sequences</code>	O mesmo que <code>pg_statio_all_sequences</code> , exceto que somente são mostradas as seqüências do sistema (Atualmente não está definida nenhuma seqüência do sistema e, portanto, esta visão está sempre vazia).
<code>pg_statio_user_sequences</code>	O mesmo que <code>pg_statio_all_sequences</code> , exceto que somente são mostradas as seqüências de usuário.

As estatísticas por índice são particularmente úteis para determinar quais índices estão sendo utilizados e quão efetivos são.

As visões `pg_statio_` são úteis, principalmente, para determinar a efetividade do `cache` do `buffer`. Quando o número de leituras físicas no disco é muito menor do que o número de acertos no `buffer`, então o `cache` está respondendo à



maioria das solicitações de leitura, evitando chamadas ao núcleo. Entretanto, estas estatísticas não fornecem toda a história: devido à forma como o PostgreSQL trata a E/S em disco, dados que não estão no `cache` do `buffer` do PostgreSQL podem estar no `cache` de E/S do núcleo e, portanto, podem ser lidos sem que haja necessidade de uma leitura física. Os usuários interessados em obter informações mais detalhadas sobre o comportamento de E/S do PostgreSQL, são aconselhados a utilizar o coletor de estatísticas do PostgreSQL em combinação com os utilitários do sistema operacional que permitem analisar o tratamento da E/S pelo núcleo.

Podem ser criadas outras formas de ver as estatísticas, escrevendo consultas que utilizam as mesmas funções subjacentes de acesso às estatísticas utilizadas pelas visões padrão. Estas funções estão listadas na Tabela 23-2. As funções de acesso por banco de dados, recebem como argumento o OID do banco de dados que identifica para qual banco de dados é o relatório. As funções por tabela e por índice recebem o OID da tabela ou do índice, respectivamente (Deve ser observado que somente podem ser vistos por estas funções as tabelas e índices presentes no banco de dados corrente). As funções de acesso por processo servidor recebem o número de ID do processo servidor, que varia de um ao número de processos servidor ativos no momento.

**Tabela 23-2. Funções de acesso às estatísticas**

Função	Tipo retornado	Descrição
<code>pg_stat_get_db_numbackends(oid)</code>	integer	Número de processos servidor ativos conectados ao banco de dados
<code>pg_stat_get_db_xact_commit(oid)</code>	bigint	Transações efetivadas no banco de dados
<code>pg_stat_get_db_xact_rollback(oid)</code>	bigint	Transações canceladas no banco de dados
<code>pg_stat_get_db_blocks_fetched(oid)</code>	bigint	Número de solicitações de busca de blocos de disco para o banco de dados
<code>pg_stat_get_db_blocks_hit(oid)</code>	bigint	Número de solicitações de busca de blocos de disco para o banco de dados encontradas no <code>cache</code>
<code>pg_stat_get_numscans(oid)</code>	bigint	Número de varreduras sequenciais realizadas quando o argumento é uma tabela, ou o número de varreduras de índice quando o argumento é um índice
<code>pg_stat_get_tuples_returned(oid)</code>	bigint	Número de linhas lidas por varreduras sequenciais quando o argumento é uma tabela, ou o número de linhas do índice lidas quando o argumento é um índice
<code>pg_stat_get_tuples_fetched(oid)</code>	bigint	Número de linhas válidas (não expiradas) da tabela buscadas por varreduras sequenciais quando o argumento é uma tabela, ou buscadas por varreduras de índice, utilizando este índice, quando o argumento é um índice
<code>pg_stat_get_tuples_inserted(oid)</code>	bigint	Número de linhas inseridas na tabela
<code>pg_stat_get_tuples_updated(oid)</code>	bigint	Número de linhas atualizadas na tabela
<code>pg_stat_get_tuples_deleted(oid)</code>	bigint	Número de linhas excluídas da tabela
<code>pg_stat_get_blocks_fetched(oid)</code>	bigint	Número de solicitações de busca de bloco de disco para a tabela ou índice
<code>pg_stat_get_blocks_hit(oid)</code>	bigint	Número de solicitações de busca de bloco de disco encontradas no <code>cache</code> para a tabela ou o índice

Função	Tipo retornado	Descrição
<code>pg_stat_get_backend_idset()</code>	conjunto de integer	Conjunto de IDs de processos servidor ativos no momento (de 1 ao número de processos servidor ativos). Veja o exemplo de utilização no texto.
<code>pg_backend_pid()</code>	integer	ID de processo do processo servidor conectado à sessão corrente
<code>pg_stat_get_backend_pid(integer)</code>	integer	ID de processo do processo servidor especificado
<code>pg_stat_get_backend_dbid(integer)</code>	oid	ID de banco de dados do processo servidor especificado
<code>pg_stat_get_backend_userid(integer)</code>	oid	ID de usuário do processo servidor especificado
<code>pg_stat_get_backend_activity(integer)</code>	text	Comando ativo do processo servidor especificado (nulo se o usuário corrente não for um superusuário nem o mesmo usuário da sessão sendo consultada, ou se <code>stats_command_string</code> não estiver habilitado)
<code>pg_stat_get_backend_activity_start(integer)</code>	timestamp with time zone	A hora em que o comando executando no momento, no processo servidor especificado, começou (nulo se o usuário corrente não for um superusuário nem o mesmo usuário da sessão sendo consultada, ou se <code>stats_command_string</code> não estiver habilitado)
<code>pg_stat_reset()</code>	boolean	Reinicia todas as estatísticas atualmente coletadas

**Nota:** `pg_stat_get_db_blocks_fetched` menos `pg_stat_get_db_blocks_hit` fornece o número de chamadas à função `read()` do núcleo feitas para a tabela, índice ou banco de dados; mas o número verdadeiro de leituras físicas é geralmente menor por causa da buferização no nível do núcleo.

A função `pg_stat_get_backend_idset` fornece uma maneira conveniente de gerar uma linha para cada processo servidor ativo. Por exemplo, para mostrar o PID e o comando corrente de todos os processos servidor:

```
SELECT pg_stat_get_backend_pid(s.backendid) AS procpid,
       pg_stat_get_backend_activity(s.backendid) AS current_query
FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

### 23.3. Ver os bloqueios

Outra ferramenta útil para monitorar a atividade do banco de dados é a tabela do sistema `pg_locks`. Esta tabela permite ao administrador do banco de dados ver informações sobre os bloqueios ativos no gerenciador de bloqueios. Por exemplo, esta funcionalidade pode ser utilizada para:

- Ver todos os bloqueios ativos no momento, todos os bloqueios nas relações em um determinado banco de dados, todos os bloqueios em uma determinada relação, ou todos os bloqueios mantidos por uma determinada sessão do PostgreSQL.
- Determinar a relação no banco de dados corrente com mais bloqueios não concedidos (que pode ser uma origem de contenção entre clientes do banco de dados).
- Determinar o efeito da contenção de bloqueio sobre o desempenho global do banco de dados, assim como até que ponto a contenção varia com o tráfego global do banco de dados.

Os detalhes da visão `pg_locks` são mostrados na Seção 41.33. Para obter informações adicionais sobre bloqueio e gerenciamento de simultaneidade no PostgreSQL, consulte o Capítulo 12.

## Notas

1. A conexão com um outro banco de dados, ou por outro usuário, fecha a conexão existente e cria uma nova conexão com outro identificador de processo. (N. do T.)
2. O `pgpool` é um servidor de `pool` de conexões para o PostgreSQL, que executa entre os clientes do PostgreSQL e os processos servidor. O cliente do PostgreSQL se conecta ao `pgpool` como se este fosse um servidor PostgreSQL padrão. Quando se usa o `pgpool` é monitorado o `pool`, e não um determinado cliente. (N. do T.)

# Capítulo 24. Monitoramento da utilização de disco

Este capítulo discute como monitorar a utilização de disco por um sistema de banco de dados PostgreSQL.

## 24.1. Determinação da utilização de disco

Cada tabela possui um arquivo em disco `heap` primário, onde a maior parte dos dados são armazenados. Se a tabela possuir alguma coluna com valor potencialmente longo, também existirá um arquivo `TOAST` associado à tabela, utilizado para armazenar os valores muito longos para caber confortavelmente na tabela principal (consulte a Seção 49.2). Haverá um índice para a tabela `TOAST`, caso esta esteja presente. Também podem haver índices associados à tabela base. Cada tabela e índice é armazenado em um arquivo em disco separado — possivelmente mais de um arquivo, se o arquivo exceder um gigabyte. As convenções para atribuir nomes a estes arquivos estão descritas na Seção 49.1.

O espaço em disco pode ser monitorado a partir de três lugares: do `psql` utilizando as informações do `VACUUM`, do `psql` utilizando as ferramentas presentes em `contrib/dbsize`, e da linha de comando utilizando as ferramentas presentes em `contrib/oid2name`. Utilizando o `psql` em um banco de dados onde o comando `VACUUM` ou `ANALYZE` foi executado recentemente, podem ser efetuadas consultas para ver a utilização do espaço em disco de qualquer tabela:

```
cep=# VACUUM ANALYZE;  
VACUUM
```

```
cep=# SELECT relname, relfilenode, relpages  
cep=# FROM pg_class  
cep=# WHERE relname LIKE 'tbl_cep_%'  
cep=# ORDER BY relname;
```

relname	relfilenode	relpages
tbl_cep_ac	17145	12
tbl_cep_al	17148	63
tbl_cep_am	17151	91
tbl_cep_ap	17154	9
tbl_cep_ba	17157	260
tbl_cep_ce	17163	216
tbl_cep_df	17166	289
tbl_cep_es	17169	218
tbl_cep_especial	17229	213
tbl_cep_go	17172	316
tbl_cep_ma	17175	67
tbl_cep_mg	17178	763
tbl_cep_mrj	17181	300
tbl_cep_ms	17184	129
tbl_cep_mt	17233	112
tbl_cep_pa	17187	139
tbl_cep_pb	17190	105
tbl_cep_pe	17193	450
tbl_cep_pi	17196	41
tbl_cep_pr	17199	482
tbl_cep_rj	17202	842
tbl_cep_rn	17205	94
tbl_cep_ro	17208	34
tbl_cep_rr	17211	13
tbl_cep_rs	17214	401
tbl_cep_sc	17217	245
tbl_cep_se	17220	34
tbl_cep_sp	17223	2755
tbl_cep_to	17226	46
tbl_cep_uf	17231	1

(30 linhas)

Cada página possui, normalmente, 8 kilobytes (Lembre-se, relpages somente é atualizado por VACUUM, ANALYZE e uns poucos comandos de DDL como CREATE INDEX). O valor de relfilenode possui interesse caso se deseje examinar diretamente o arquivo em disco da tabela.

Para ver o espaço utilizado pelas tabelas TOAST deve ser utilizada uma consulta como a mostrada abaixo:

```
-- Ver as tabelas TOAST da tabela pg_rewrite
cep=# SELECT relname, relpages
cep-# FROM pg_class,
cep-# (SELECT reltoastrelid FROM pg_class
cep-# WHERE relname = 'pg_rewrite') ss
cep-# WHERE oid = ss.reltoastrelid
cep-# OR oid = (SELECT reltoastidxid FROM pg_class
cep-# WHERE oid = ss.reltoastrelid)
cep-# ORDER BY relname;
```

relname	relpages
pg_toast_16410	14
pg_toast_16410_index	2

(2 linhas)

Os tamanhos dos índices também podem ser facilmente exibidos:

```
cep=# ALTER TABLE tbl_cep_sp ADD PRIMARY KEY (cep);
NOTICE: ALTER TABLE / ADD PRIMARY KEY will create implicit index "tbl_cep_sp_pkey"
        for table "tbl_cep_sp"
ALTER TABLE
```

```
cep=# SELECT c2.relname, c2.relpages
cep-# FROM pg_class c, pg_class c2, pg_index i
cep-# WHERE c.relname = 'tbl_cep_sp'
cep-# AND c.oid = i.indrelid
cep-# AND c2.oid = i.indexrelid
cep-# ORDER BY c2.relname;
```

relname	relpages
tbl_cep_sp_pkey	619

(1 linha)

```
-- Verificar se relpages foi atualizada na criação do índice
```

```
cep=# VACUUM FULL ANALYZE;
VACUUM
```

```
cep=# SELECT c2.relname, c2.relpages
cep-# FROM pg_class c, pg_class c2, pg_index i
cep-# WHERE c.relname = 'tbl_cep_sp'
cep-# AND c.oid = i.indrelid
cep-# AND c2.oid = i.indexrelid
cep-# ORDER BY c2.relname;
```

relname	relpages
tbl_cep_sp_pkey	619

(1 linha)

```
-- O valor de relpages não mudou após VACUUM FULL ANALYZE
```

Utilizando a consulta abaixo, é fácil descobrir quais são as quatro maiores tabelas e índices:

```

cep=# SELECT relname, relpages
cep=#      FROM pg_class
cep=#      ORDER BY relpages DESC
cep=#      LIMIT 4;

```

```

      relname      | relpages
-----+-----
tbl_cep_sp        |      2754
tbl_cep_rj        |        841
tbl_cep_mg        |        762
tbl_cep_sp_pkey   |        619
(4 linhas)

```

-- Como pode ser visto, VACUUM FULL ANALYZE diminuiu uma página de tbl\_cep\_sp após VACUUM ANALYZE (2755 para 2754).

Calcular o número de páginas da tabela tbl\_cep\_sp através das informações do sistema de arquivos, utilizando o nome de arquivo presente em relfilenode (N. do T.):

```

$ locate 17223 | grep /var/lib/pgsql/data/base/
/var/lib/pgsql/data/base/17142/17223
$ ls -l /var/lib/pgsql/data/base/17142/17223
-rw----- 1 postgres postgres 22560768 Jul 28 07:44 /var/lib/pgsql/data/base/17142/17223
$ expr 22560768 / 8 / 1024
2754

```

Como cada página possui 8 kilobytes, dividindo-se o tamanho do arquivo igual a 22.560.768 bytes por 8 e por 1024 chega-se a um número de páginas igual a 2754, o mesmo informado pela consulta.

A ferramenta contrib/dbsize carrega funções no banco de dados que permitem descobrir o tamanho da tabela ou do banco de dados a partir do psql sem necessidade do VACUUM ou do ANALYZE.

Para mostrar a utilização de disco também pode ser utilizada a ferramenta contrib/oid2name. Veja exemplo em README.oid2name neste diretório. Inclui um script que mostra utilização de disco para cada banco de dados.

## 24.2. Falha de disco cheio

A tarefa mais importante de monitoramento de disco do administrador de banco de dados é garantir que o disco não vai ficar cheio. Um disco de dados cheio não resulta em corrupção dos dados, mas pode impedir que ocorram atividades úteis. Se o disco que contém os arquivos do WAL ficar cheio, pode acontecer do servidor de banco de dados entrar na condição de pânico e encerrar a execução.

Se não for possível liberar espaço adicional no disco removendo outros arquivos, podem ser movidos alguns arquivos do banco de dados para outros sistemas de arquivos utilizando espaços de tabelas. Para obter mais informações sobre este assunto deve ser consultada a Seção 18.6.

**Dica:** Alguns sistemas de arquivos têm desempenho degradado quando estão praticamente cheios, portanto não se deve esperar o disco ficar cheio para agir.

Se o sistema possuir cotas de disco por usuário, então naturalmente o banco de dados estará sujeito a cota atribuída para o usuário sob o qual o sistema de banco de dados executa. Exceder a cota ocasiona os mesmos malefícios de ficar totalmente sem espaço.

# Capítulo 25. Registro prévio da escrita (WAL)

O *registro prévio da escrita* (WAL = *write ahead logging*) é uma abordagem padrão para registrar transações. A descrição detalhada pode ser encontrada na maioria (se não em todos) os livros sobre processamento de transação. Em poucas palavras, o conceito central do WAL é que as alterações nos arquivos de dados (onde as tabelas e os índices residem) devem ser escritas somente após estas alterações terem sido registradas, ou seja, quando os registros que descrevem as alterações tiverem sido descarregados em um meio de armazenamento permanente. Se este procedimento for seguido, não será necessário descarregar as páginas de dados no disco a cada efetivação de transação, porque se sabe que no evento de uma queda será possível recuperar o banco de dados utilizando o registro: todas as alterações que não foram aplicadas às páginas de dados são refeitas a partir dos registros (isto é a recuperação de rolar para a frente, *roll-forward*, também conhecida como REDO),

## 25.1. Benefícios do WAL

O primeiro grande benefício da utilização do WAL é a redução significativa do número de escritas em disco, uma vez que na hora em que a transação é efetivada somente precisa ser descarregado em disco o arquivo de registro, em vez de todos os arquivos de dados modificados pela transação. Em ambiente multiusuário, a efetivação de várias transações pode ser feita através de um único `fsync()` do arquivo de registro. Além disso, o arquivo de registro é escrito sequencialmente e, portanto, o custo de sincronizar o registro é muito menor do que o custo de descarregar as páginas de dados. Isto é especialmente verdade em servidores tratando muitas transações pequenas afetando partes diferentes do armazenamento de dados.

O benefício seguinte é a consistência das páginas de dados. A verdade é que antes do WAL o PostgreSQL nunca foi capaz de garantir a consistência no caso de uma queda. Antes do WAL, qualquer queda durante a escrita poderia resultar em:

1. linhas de índice apontando para linhas inexistentes da tabela
2. perda de linhas de índice nas operações de quebra de página (`split`)
3. conteúdo da página da tabela ou do índice totalmente corrompido, por causa das páginas de dados parcialmente escritas

Os problemas com os índices (problemas 1 e 2) possivelmente poderiam ter sido resolvidos através de chamadas adicionais à função `fsync()`, mas não é óbvio como tratar o último caso sem o WAL; se for necessário, o WAL salva todo o conteúdo da página de dados no registro, para garantir a consistência da página na recuperação após a queda.

Por fim, o WAL permite que seja feita cópia de segurança em linha e recuperação para um ponto no tempo, conforme descrito na Seção 22.3. Fazendo cópia dos arquivos de segmento do WAL pode-se retornar para qualquer instante no tempo coberto pelos registros do WAL: simplesmente se instala uma versão anterior da cópia de segurança física do banco de dados, e se refaz o WAL até o ponto desejado no tempo. Além disso, a cópia de segurança física não precisa ser um instantâneo do estado do banco de dados — se a cópia for realizada durante um período de tempo, quando o WAL for refeito para este período de tempo da cópia serão corrigidas todas as inconsistências internas.

## 25.2. Configuração do WAL

Existem diversos parâmetros de configuração relacionados com o WAL que afetam o desempenho do banco de dados. Esta seção explica como defini-los. Para obter detalhes gerais sobre como definir parâmetros de configuração para o servidor deve ser consultada a Seção 16.4.

Os *pontos de controle* (*checkpoints*) são pontos na sequência de transações onde se garante que os arquivos de dados foram atualizados com todas as informações registradas antes deste ponto. No momento do ponto de controle, todas as páginas de dados sujas (*dirty*) são descarregadas no disco, e é escrito um registro especial de ponto de controle no arquivo de segmento do WAL. Como resultado, no caso de uma queda o procedimento de recuperação sabe a partir de qual ponto do WAL (chamado de registro de refazer (REDO)) deve começar a operação de refazer, uma vez que todas as mudanças feitas nos arquivos de dados anteriores a este ponto já se encontram gravadas em disco. Após ter sido feito o ponto de controle, nenhum dos arquivos de segmento do WAL, escritos antes do registro de refazer, continua sendo necessário, podendo ser reciclados ou removidos (Quando está sendo feita cópia dos arquivos de segmento do WAL, os arquivos de segmento devem ser copiados antes de serem reciclados ou removidos).

O processo de escrita em segundo plano do servidor realiza, automaticamente, um ponto de controle de tempo em tempo. Um ponto de controle é realizado a cada `checkpoint_segments` do WAL, ou a cada `checkpoint_timeout` segundos, o que ocorrer primeiro. As definições padrão são 3 segmentos e 300 segundos, respectivamente. Também é possível obrigar a realização de um ponto de controle utilizando o comando SQL `CHECKPOINT`.

Reduzir `checkpoint_segments` e/ou `checkpoint_timeout` faz os pontos de controle serem feitos com maior frequência, permitindo uma recuperação mais rápida após a queda (uma vez que haverá menos trabalho para ser feito). Entretanto, deve haver um balanço entre isto e o aumento de custo causado pela descarga das páginas sujas com maior frequência. Além disso, após cada ponto de controle, para garantir a consistência das páginas de dados, a primeira modificação feita em uma página de dados ocasiona o registro de todo o conteúdo desta página. Por isso, um intervalo de ponto de controle menor aumenta o volume de saída para o WAL, negando parcialmente o objetivo de utilizar um intervalo menor e, em qualquer caso, causando mais E/S em disco.

Os pontos de controle são muito dispendiosos, primeiro porque requerem a escrita de todos os `buffers` sujos correntes, e depois porque resultam em um tráfego adicional subsequente para o WAL, conforme visto acima. Portanto, é aconselhável definir os parâmetros do ponto de controle altos o suficiente para que não ocorram com muita frequência. Como uma verificação de sanidade simples para os parâmetros de ponto de controle, pode ser definido o parâmetro `checkpoint_warning`. Se os pontos de controle ocorrerem antes de `checkpoint_warning` segundos, será gerada uma mensagem para o log do servidor recomendando o aumento de `checkpoint_segments`. Uma aparição ocasional desta mensagem não é motivo de alarme, mas se aparecer com frequência então os parâmetros de controle de ponto de verificação devem ser aumentados.

Há pelo menos um arquivo de segmento e, normalmente, não mais de  $2 * \text{checkpoint\_segments} + 1$  arquivos de segmento do WAL. Normalmente cada arquivo de segmento possui o tamanho de 16 MB (embora este tamanho possa ser alterado na construção do servidor). Isto pode ser utilizado para estimar a necessidade de espaço do WAL. Normalmente, quando os arquivos de segmento do WAL antigos não são mais necessários, estes são reciclados (os nome são mudados para se tornarem os próximos segmentos da sequência numerada). Se, por causa de um pico de pouca duração da taxa de saída para o WAL, existirem mais de  $2 * \text{checkpoint\_segments} + 1$  arquivos de segmento, os arquivos de segmento desnecessários serão removidos em vez de reciclados até o sistema voltar a ficar abaixo deste limite.

Existem duas funções do WAL utilizadas com frequência: `LogInsert` e `LogFlush`. A função `LogInsert` é utilizada para colocar um novo registro nos `buffers` do WAL na memória compartilhada. Se não houver espaço para o novo registro, `LogInsert` terá que escrever (mover para o cache do núcleo) uns poucos `buffers` do WAL cheios. Isto não é desejado, porque `LogInsert` é utilizada em todas as modificações de baixo nível do banco de dados (por exemplo, inserção de linha), quando um bloqueio exclusivo é mantido nas páginas de dados afetadas, portanto esta operação precisa ser tão rápida quanto for possível. O que é pior, escrever os `buffers` do WAL também pode obrigar a criação de um novo arquivo de segmento, que toma mais tempo ainda. Normalmente, os `buffers` do WAL devem ser escritos e descarregados pela chamada `LogFlush` que é feita, na maioria das vezes, na hora da efetivação da transação para garantir que os registros da transação sejam descarregados no armazenamento permanente. Nos sistemas com saída para o WAL alta, as chamadas à `LogFlush` podem não ocorrer com uma frequência suficiente para evitar que `LogInsert` tenha que realizar escritas. Nestes sistemas, deve ser aumentado o número de `buffers` do WAL pela modificação do parâmetro `wal_buffers`. O número padrão de `buffers` do WAL é 8. O aumento deste valor aumenta de forma correspondente a utilização de memória compartilhada (Deve ser observado que no momento existe pouca evidência sugerindo que aumentar `wal_buffers` acima do padrão valha a pena).

O parâmetro `commit_delay` define a quantidade de microssegundos que o processo servidor vai aguardar após escrever um registro de efetivação no WAL com `LogInsert`, antes de realizar o `LogFlush`. Este retardo permite que outros processos servidor adicionem seus registros de efetivação ao WAL para que todos sejam descarregados em uma única sincronização do WAL. Não ocorre nenhum retardo quando `fsync` não está habilitado, nem se menos de outras `commit_siblings` sessões estiverem com transações ativas no momento; isto evita o retardo quando é pouco provável que outras seções efetivem em breve. Deve ser observado que na maioria das plataformas a resolução de uma solicitação de retardo é de dez milissegundos, portanto qualquer definição de `commit_delay` diferente de zero e entre 1 e 10000 microssegundos produz o mesmo efeito. Ainda não está claro quais são os melhores valores para estes parâmetros; incentiva-se que sejam feitas experiências.

O parâmetro `wal_sync_method` determina como o PostgreSQL vai fazer a solicitação ao núcleo para forçar o envio das atualizações do WAL para o disco. Todas as opções devem ser idênticas em termos de confiabilidade, mas é bem específico da plataforma qual delas é a mais rápida. Deve ser observado que este parâmetro é irrelevante se `fsync` estiver desabilitado.



Habilitar o parâmetro de configuração `wal_debug` (desde que o PostgreSQL tenha sido compilado com suporte ao mesmo) resulta em que cada chamada às funções `LogInsert` e `LogFlush` feita pelo WAL seja registrada no `log` do servidor. Esta opção poderá ser substituída por um mecanismo mais genérico no futuro.

### 25.3. Internamente

O WAL é habilitado automaticamente; não é requerida nenhuma ação por parte do administrador, exceto garantir que o espaço em disco adicional necessário para o WAL seja atendido, e que seja feito qualquer ajuste necessário (consulte a Seção 25.2).

O WAL é armazenado no diretório `pg_xlog`, sob o diretório de dados, como um conjunto de arquivos de segmento, normalmente com o tamanho de 16 MB cada. Cada segmento é dividido em páginas, normalmente de 8 kB cada. Os cabeçalhos dos registros estão descritos em `access/xlog.h`; o conteúdo do registro depende do tipo de evento que está sendo registrado. São atribuídos para nomes dos arquivos de segmento números que sempre aumentam, começando por 00000001000000000000000000. Atualmente os números não recomeçam, mas deve demorar muito tempo até que seja exaurido o estoque de números disponíveis.

Os `buffers` do WAL e estruturas de controle ficam na memória compartilhada e são tratados pelos processos servidor filhos; são protegidos por bloqueios de peso leve. A demanda por memória compartilhada é dependente do número de `buffers`. O tamanho padrão dos `buffers` do WAL é 8 `buffers` de 8 kB cada um, ou um total de 64 kB.

É vantajoso o WAL ficar localizado em um disco diferente do que ficam os arquivos de banco de dados principais. Isto pode ser obtido movendo o diretório `pg_xlog` para outro local (enquanto o servidor estiver parado, é óbvio), e criando um vínculo simbólico do local original no diretório de dados principal para o novo local.

A finalidade do WAL, garantir que a alteração seja registrada antes que as linhas do banco de dados sejam alteradas, pode ser subvertida pelos controladores de disco (`drives`) que informam ao núcleo uma escrita bem-sucedida falsa, e na verdade apenas colocam os dados no `cache` sem armazenar no disco. Numa situação como esta a queda de energia pode conduzir a uma corrupção dos dados não recuperável. Os administradores devem tentar garantir que os discos que armazenam os arquivos de segmento do WAL do PostgreSQL não fazem estes falsos relatos.

Após um ponto de controle ter sido feito e o registro descarregado, a posição do ponto de controle é salva no arquivo `pg_control`. Portanto, quando uma recuperação vai ser feita o servidor lê primeiro `pg_control`, e depois o registro de ponto de controle; em seguida realiza a operação de REDO varrendo para frente a partir da posição indicada pelo registro de ponto de controle. Como, após o ponto de controle, na primeira modificação feita em uma página de dados é salvo todo o conteúdo desta página, todas as páginas modificadas desde o último ponto de controle serão restauradas para um estado consistente.

Para tratar o caso em que o arquivo `pg_control` foi corrompido, é necessário haver suporte para a possibilidade de varrer os arquivos de segmento do WAL em sentido contrário — mais novo para o mais antigo — para encontrar o último ponto de controle. Isto ainda não foi implementado. O arquivo `pg_control` é pequeno o suficiente (menos que uma página de disco) para não estar sujeito a problemas de escrita parcial, e até o momento em que esta documentação foi escrita não haviam relatos de falhas do banco de dados devido unicamente a incapacidade de ler o arquivo `pg_control`. Portanto, embora este seja teoricamente um ponto fraco, na prática o arquivo `pg_control` não parece ser um problema.

# Capítulo 26. Testes de regressão

Os testes de regressão compõem um conjunto detalhado de testes da implementação da linguagem SQL no PostgreSQL. São testadas as operações SQL padrão, assim como as funcionalidades estendidas do PostgreSQL.

## 26.1. Execução dos testes

Os testes de regressão podem ser executados em um servidor já instalado e em funcionamento, ou utilizando uma instalação temporária dentro da árvore de construção. Além disso, existem os modos “paralelo” e “seqüencial” para execução dos testes. O modo seqüencial executa cada um dos scripts de teste por vez, enquanto o modo paralelo inicia vários processos servidor para executar grupos de teste em paralelo. Os testes em paralelo dão a confiança de que a comunicação entre processos e os bloqueios estão funcionando de forma correta. Por motivos históricos, os testes seqüenciais são geralmente executados em uma instalação existente, e o modo paralelo em uma instalação temporária, mas não há motivo técnico para ser feito assim.

Para executar os testes de regressão após construir, mas antes de instalar, deve ser digitado

**gmake check**

no diretório de nível mais alto (Ou o diretório `src/test/regress` pode ser tornado o diretório corrente e o comando executado neste diretório). Primeiro serão construídos vários arquivos auxiliares, como algumas funções de gatilho de exemplo definidas pelo usuário e, depois, executado o script condutor do teste. Ao final deve ser visto algo parecido com

```
=====
All 96 tests passed.
=====
```

ou, senão, uma nota sobre quais testes falharam. Deve ser vista a Seção 26.2 abaixo antes de assumir que “failure” representa um problema sério.

Uma vez que este método executa um servidor temporário, não funciona quando se está logado como o usuário `root` (uma vez que o servidor não inicializa sob o `root`). Se foi feita a construção como `root`, não será necessário começar tudo novamente. Em vez disto, deve ser feito com que o diretório do teste de regressão possa ser escrito por algum outro usuário, se conectar como este usuário, e recomendar os testes. Por exemplo:

```
root# chmod -R a+w src/test/regress
root# chmod -R a+w contrib/spi
root# su - joeuser
joeuser$ cd diretório_de_construção_de_nível_mais_alto
joeuser$ gmake check
```

(O único “risco de segurança” possível neste caso é a possibilidade de outro usuário alterar os resultados do teste de regressão sem que se saiba. Use o bom senso ao gerenciar permissões para usuários).

Como alternativa, os testes podem ser executados após a instalação.

Se o PostgreSQL for configurado para ser instalado em um local onde existe uma instalação mais antiga do PostgreSQL, e for executado `gmake check` antes de instalar a nova versão, pode ser que os testes falhem devido aos novos programas tentarem utilizar as bibliotecas compartilhadas já instaladas (O sintoma típico é a reclamação sobre símbolos não definidos). Se for desejado executar os testes antes de sobrescrever a instalação antiga, será necessário fazer a construção utilizando `configure --disable-rpath`. Entretanto, não se recomenda que esta opção seja utilizada na instalação final.

O teste de regressão paralelo inicia alguns processos sob o ID do usuário. Atualmente, a simultaneidade máxima são vinte scripts de teste em paralelo, o que significa sessenta processos: para cada script de teste existe um processo servidor, o `psql` e, geralmente, o processo ancestral do interpretador de comandos que chamou o `psql`. Portanto, se o sistema impõe limite para o número de processos por usuário, certifique-se que este limite é de setenta e cinco ou mais, senão podem acontecer falhas aleatórias no teste paralelo. Se não houver condição de aumentar este limite, pode ser diminuído o grau de paralelismo definindo o parâmetro `MAX_CONNECTIONS`. Por exemplo,

```
gmake MAX_CONNECTIONS=10 check
```

não executa mais de dez testes ao mesmo tempo.

Em alguns sistemas o interpretador de comandos padrão compatível com o Bourne (`/bin/sh`) se confunde ao gerenciar tantos processos descendentes em paralelo, podendo fazer com que o teste paralelo trave ou falhe. Neste caso, deve ser especificado na linha de comandos um interpretador de comandos compatível com o Bourne diferente como, por exemplo:

```
gmake SHELL=/bin/ksh check
```

Se não estiver disponível nenhum interpretador de comandos que não apresente este problema, então este problema poderá ser contornado limitando o número de conexões, conforme mostrado acima.

Para executar os testes após a instalação (consulte o Capítulo 14), deve ser inicializada a área de dados e ativado o servidor e depois, conforme explicado no Capítulo 16, deve ser digitado:

```
gmake installcheck
```

ou para o teste em paralelo

```
gmake installcheck-parallel
```

Os testes esperam fazer contato com o servidor no hospedeiro local no número de porta padrão, a menos que esteja especificado o contrário nas variáveis de ambiente `PGHOST` e `PGPORT`.

## 26.2. Avaliação dos testes

Algumas instalações do PostgreSQL, devidamente instaladas e inteiramente funcionais, podem “falhar” em alguns testes de regressão por causa de algumas particularidades específicas da plataforma, como a representação diferente do ponto flutuante ou do suporte à zona horária. Atualmente a avaliação dos testes é feita simplesmente usando o programa `diff`, para comparar os resultados obtidos pelos testes com os resultados produzidos no sistema de referência e, portanto, os testes são sensíveis a pequenas diferenças entre sistemas. Quando for relatado que o teste “falhou”, sempre deve ser examinada a diferença entre o resultado esperado e o resultado obtido; pode-se descobrir que as diferenças não são significativas. Ainda assim, são feitos esforços para manter os arquivos de referência idênticos entre todas as plataformas suportadas, portanto deve-se esperar que todos os testes sejam bem-sucedidos.

As saídas produzidas pelos testes de regressão ficam nos arquivos do diretório `src/test/regress/results`. Os scripts dos testes utilizam o programa `diff` para comparar cada arquivo de saída produzido com a saída de referência armazenada no diretório `src/test/regress/expected`. Todas as diferenças são salvas em `src/test/regress/regression.diffs` para poderem ser inspecionadas (ou pode ser executado o programa `diff` diretamente, se for preferido).

### 26.2.1. Diferenças nas mensagens de erro

Alguns testes de regressão envolvem, intencionalmente, valores de entrada inválidos. As mensagens de erro podem ser originadas pelo código do PostgreSQL, ou pelas rotinas do sistema da plataforma hospedeira. No último caso, as mensagens podem variar entre plataformas, mas devem conter informações semelhantes. Estas diferenças nas mensagens resultam em testes de regressão que “falham”, mas que podem ser validados por inspeção.

### 26.2.2. Diferenças no idioma

Se os testes forem executados em um servidor já instalado, que foi inicializado com uma ordem de classificação das cadeias de caracteres (`LC_COLLATE`) em um idioma diferente de C, então podem haver diferenças por causa da ordem de classificação e falhas de continuação. O conjunto de testes de regressão é configurado para tratar este problema mediante arquivos de resultado alternativos, que juntos podem tratar um grande número de idiomas. Por exemplo, para o teste `char` o arquivo esperado `char.out` trata os idiomas C e POSIX, e o arquivo `char_1.out` trata vários outros idiomas. O condutor do teste de regressão pega, automaticamente, o melhor arquivo para fazer a comparação quando verifica se foi bem-sucedido, e para computar as diferenças da falha (Isto significa que os testes de regressão não conseguem detectar se os resultados são apropriados para o idioma configurado. Os testes simplesmente pegam o arquivo de resultado que melhor se adequar).

Se por alguma razão os arquivos esperados existentes não incluírem algum idioma, é possível adicionar um novo arquivo. O esquema de nomes é `nomedoteste_dígito.out`. O dígito utilizado não tem importância. Lembre-se que o condutor do teste de regressão considera todos os arquivos deste tipo como sendo resultados de teste igualmente válidos. Se os resultados do teste forem específicos para alguma plataforma, em vez dessa abordagem deve ser utilizada a técnica descrita na Seção 26.3.

### 26.2.3. Diferenças na data e hora

Um poucas consultas do teste `horology`<sup>1</sup> falham se forem executadas no dia de início ou de fim do horário de verão, ou no dia seguinte aos mesmos. Estas consultas esperam que os intervalos entre a meia-noite do dia anterior, a meia noite do dia, e a meia-noite do dia seguinte, sejam de exatamente vinte e quatro horas — o que não acontece se o início ou o fim do horário de verão ocorrer neste intervalo.

**Nota:** Uma vez que são utilizadas as regras de horário de verão (`daylight-saving time`) dos EUA, este problema sempre ocorre no primeiro domingo de abril, no último domingo de outubro, e nas segundas-feiras seguintes, não importando quando o horário de verão começa ou termina onde se vive. Deve ser observado, também, que este problema aparece ou desaparece à meia-noite do Horário do Pacífico (UTC-7 ou UTC-8), e não à meia-noite do horário local. Portanto, a falha pode ocorrer no sábado ou durar até terça-feira dependendo de onde se vive.

A maior parte dos resultados de data e hora são dependentes da zona horária do ambiente. Os arquivos de referência são gerados para a zona horária `PST8PDT` (Berkeley, Califórnia), havendo falhas aparentes se os testes não forem executados com esta definição de zona horária. O condutor do teste de regressão define a variável de ambiente `PGTZ` como `PST8PDT`, o que normalmente garante resultados apropriados.

### 26.2.4. Diferenças no ponto flutuante

Alguns testes incluem cálculos com números de ponto flutuante de 64 bits (`double precision`) a partir de colunas das tabelas. Foram observadas diferenças nos resultados quando estão envolvidas funções matemáticas aplicadas a colunas do tipo `double precision`. Os testes `float8` e `geometry` são particularmente propensos a apresentar pequenas diferenças entre plataformas, ou devido a opções diferentes de otimização do compilador. São necessárias comparações utilizando o olho humano para determinar se estas diferenças, que geralmente estão dez casas à direita do ponto decimal, são significativas.

Alguns sistemas mostram menos zero como `-0`, enquanto outros mostram simplesmente `0`.

Alguns sistemas sinalizam erros das funções `pow()` e `exp()` de forma diferente da esperada pelo código corrente do PostgreSQL.

### 26.2.5. Diferenças na ordem das linhas

Podem ser encontradas diferenças onde as mesmas linhas são mostradas em uma ordem diferente da que aparece no arquivo de comparação. Na maior parte das vezes isto não é, a rigor, um erro. Os scripts de teste de regressão, em sua maioria, não são tão refinados a ponto de utilizar a cláusula `ORDER BY` em todos os comandos `SELECT` e, portanto, a ordem das linhas do resultado não é bem definida, de acordo com o texto de especificação do padrão SQL. Na prática, uma vez que se está examinando as mesmas consultas sendo executadas nos mesmos dados pelo mesmo programa, geralmente são obtidos resultados na mesma ordem em todas as plataformas e, por isso, a ausência do `ORDER BY` não se torna um problema. Entretanto, algumas consultas apresentam diferenças na ordem das linhas entre plataformas (Diferenças na ordem das linhas também podem ser ocasionadas pela definição de um idioma diferente de C).

Portanto, se houver diferença na ordem das linhas isto não é algo com que devamos nos preocupar, a menos que a consulta possua uma cláusula `ORDER BY` que esteja sendo violada. Mas, por favor, informe de qualquer forma para que possamos adicionar a cláusula `ORDER BY` a esta consulta em particular e, com isso, eliminar a falsa “falha” nas próximas versões.

Deve-se estar querendo saber porque não se ordena explicitamente todas as consultas dos testes de regressão imediatamente, para acabar com este problema de uma vez e para sempre. A razão é que isto torna os testes de regressão menos úteis, e não mais úteis, porque vão tender a utilizar tipos de plano de consulta que produzem resultados ordenados, prejudicando os planos que não o fazem.

### 26.2.6. O teste “random”

O script do teste `random` tem por objetivo produzir resultados aleatórios. Raramente isto faz com que o teste de regressão `random` falhe. Ao ser digitado

```
diff results/random.out expected/random.out
```

deve ser produzida somente uma, ou umas poucas linhas diferentes. Não é necessário se preocupar com isto, a menos que o teste falhe repetidamente.

## 26.3. Arquivos de comparação específicos de plataformas

Como alguns testes produzem resultados inerentes a uma determinada plataforma, é disponibilizada uma maneira de fornecer arquivos de comparação de resultados específicos para a plataforma. Com frequência a mesma discrepância se aplica a várias plataformas; em vez de fornecer arquivos de comparação distintos para todas as plataformas, existe um arquivo de mapeamento que define o arquivo de comparação a ser utilizado. Portanto, para eliminar falsas “falhas” nos testes para uma determinada plataforma, deve ser escolhido ou desenvolvido um arquivo de resultado alternativo, e depois adicionada uma linha no arquivo de mapeamento, que é o `src/test/regress/resultmap`.

Toda linha do arquivo de mapeamento possui a forma:

```
nome_do_teste/padrão_de_plataforma=nome_do_arquivo_de_comparação
```

O nome do teste é simplesmente o nome do módulo de teste de regressão específico. O padrão de plataforma é um padrão no estilo da ferramenta Unix `expr` (ou seja, uma expressão regular com uma âncora `^` implícita no início). Este padrão é comparado com o nome da plataforma conforme exibido por `config.guess`, seguido por `:gcc` ou `:cc`, dependendo se for utilizado o compilador GNU ou o compilador nativo do sistema (nos sistemas onde há diferença). O nome do arquivo de comparação é o nome do arquivo de comparação de resultado substituto.

Por exemplo: alguns sistemas interpretam valores de ponto flutuante muito pequenos como zero, em vez de informar um erro de `underflow`. Isto causa algumas pequenas diferenças no teste de regressão `float8`. Por isso é fornecido um arquivo de comparação alternativo, `float8-small-is-zero.out`, que inclui os resultados esperados nestes sistemas. Para silenciar as mensagens falsas de “falha” nas plataformas OpenBSD, o arquivo `resultmap` inclui

```
float8/i.86-.*-openbsd=float8-small-is-zero
```

que dispara em toda máquina para a qual a saída de `config.guess` corresponde a `i.86-.*-openbsd`. Outras linhas no arquivo `resultmap` selecionam arquivos de comparação alternativos para outras plataformas conforme apropriado.

## Notas

1. *horology* — A ciência da medição do tempo, ou os princípios e a arte da construção de instrumentos para medir e indicar porções do tempo, como relógios, cronômetros, etc. Webster's Revised Unabridged Dictionary (1913) (N. do T.)

## IV. Interfaces cliente

Esta parte descreve as interfaces de programação cliente distribuídas com o PostgreSQL. Cada um destes capítulos pode ser lido de forma independente. Deve ser observado que existem muitas outras interfaces de programação para programas cliente distribuídas em separado, cada uma contendo sua própria documentação (O Apêndice H lista algumas das mais populares). Os leitores desta parte devem estar familiarizados com os comandos SQL utilizados para manipular e consultar os bancos de dados (consulte a Parte II) e, naturalmente, com a linguagem de programação utilizada pela interface.

# Capítulo 27. libpq - Biblioteca C

A libpq é a interface do programador de aplicativo C com o PostgreSQL. A libpq é um conjunto de funções de biblioteca que permitem os programas clientes passar comandos para o servidor PostgreSQL, e receber os resultados destes comandos.

A libpq também é o mecanismo subjacente de várias outras interfaces de aplicativo do PostgreSQL, incluindo as escritas para C++, Perl, Python, Tcl e ECPG. Portanto, alguns aspectos do comportamento da libpq são importantes para aqueles que utilizam um destes pacotes. Em particular, a Seção 27.11, a Seção 27.12 e a Seção 27.13 descrevem o comportamento visível aos usuários de qualquer aplicativo que utiliza a libpq.

No final deste capítulo são incluídos alguns programas curtos (Seção 27.16) para mostrar como se escreve programas que utilizam a libpq. Também existem vários exemplos completos de aplicativos libpq no diretório `src/test/examples` da distribuição do código fonte.

Os programas cliente que utilizam a libpq devem incluir o arquivo de cabeçalho `libpq-fe.h`, e devem ligar com a biblioteca libpq.

## 27.1. Funções de controle da conexão com o banco de dados

As funções a seguir lidam com o estabelecimento da conexão com o servidor PostgreSQL. Um programa aplicativo pode manter várias conexões abertas ao mesmo tempo (um motivo para isso ser feito é o acesso a mais de um banco de dados). Cada conexão é representada por um objeto `PGconn`, obtido a partir da função `PQconnectdb` ou `PQsetdbLogin`. Deve ser observado que estas funções sempre retornam um ponteiro de objeto não nulo, a menos que não haja memória suficiente para alocar o objeto `PGconn`. Antes de enviar comandos pelo objeto de conexão, deve ser chamada a função `PQstatus` para verificar se a conexão foi bem-sucedida.

`PQconnectdb`

Estabelece uma nova conexão com o servidor de banco de dados.

```
PGconn *PQconnectdb(const char *conninfo);
```

Esta função abre uma nova conexão com o servidor de banco de dados utilizando os parâmetros presentes da cadeia de caracteres `conninfo`. Diferentemente de `PQsetdbLogin` mostrada abaixo, o conjunto de parâmetros pode ser estendido sem mudar a assinatura da função, portanto deve-se dar preferência na programação de novos aplicativos ao uso desta função (ou de suas análogas não bloqueantes `PQconnectStart` e `PQconnectPoll`).

A cadeia de caracteres passada pode estar vazia, caso em que será utilizado o valor padrão de todos os parâmetros, ou pode conter a definição de um ou mais parâmetros separados por espaço em branco. Cada definição de parâmetro tem a forma `palavra_chave = valor`. Os espaços em torno do sinal de igual são opcionais. Para escrever um valor vazio, ou um valor contendo espaços, o valor deve ser colocado entre apóstrofes como, por exemplo, `palavra_chave = 'um valor'`. Apóstrofes e contra-barras dentro do valor devem receber escape de contra-barra, ou seja, `\'` e `\\`.

As palavras chave de parâmetro reconhecidas no momento são:

`host`

Nome do hospedeiro para se conectar. Se o nome começar por uma barra, especifica a comunicação com um domínio-Unix, em vez de uma comunicação TCP/IP; neste caso, valor é o nome do diretório onde o arquivo de soquete será armazenado. O comportamento padrão quando não se especifica o parâmetro `host` é conectar a um soquete do domínio-Unix no diretório `/tmp` (ou qualquer que tenha sido o diretório de soquete especificado quando o PostgreSQL foi construído). Nas máquinas sem soquete de domínio-Unix, o padrão é conectar ao `localhost`.

`hostaddr`

Endereço numérico de IP do hospedeiro a ser feita a conexão. Deve estar no formato padrão de endereço IPv4 como, por exemplo, `172.28.40.9`. Havendo suporte a IPv6 na máquina, então estes endereços também podem ser utilizados. Quando se especifica para este parâmetro uma cadeia de caracteres não vazia, é sempre utilizada uma comunicação TCP/IP.

A utilização de `hostaddr` em vez de `host` permite o aplicativo evitar a procura do hospedeiro pelo nome, que pode ser importante nos aplicativos com restrição de tempo. Entretanto, a autenticação Kerberos requer o nome do hospedeiro. Portanto, o que vem a seguir se aplica: Quando se especifica `host` e não se especifica `hostaddr`, ocorre a procura do hospedeiro pelo nome. Quando se especifica `hostaddr` e não se especifica `host`, o valor de `hostaddr` fornece o endereço remoto. Quando se utiliza o Kerberos, ocorre uma procura pelo nome reversa, para obter o nome do hospedeiro para o Kerberos. Quando se especifica tanto `host` quanto `hostaddr`, o valor de `hostaddr` fornece o endereço remoto; o valor de `host` é ignorado, a menos que se utilize o Kerberos, quando o valor é utilizado para autenticação pelo Kerberos (Deve ser observado que a autenticação não deverá ser bem-sucedida se for passado para a libpq um nome de hospedeiro que não corresponde ao nome da máquina em `hostaddr`). Além disso, é utilizado `host` para identificar a conexão, em vez de `hostaddr`, no arquivo `~/.pgpass` (consulte a Seção 27.12).

Se não for especificado o nome do hospedeiro, nem o endereço do hospedeiro, a libpq faz a conexão usando um soquete do domínio-Unix local; nas máquinas sem soquetes do domínio-Unix é feita uma tentativa de conexão com `localhost`.

`port`

Número da porta para se conectar no hospedeiro servidor, ou a extensão do nome do arquivo de soquete para as conexões com o domínio-Unix.

`dbname`

O nome do banco de dados. Por padrão o mesmo nome do usuário.

`user`

O nome do usuário do PostgreSQL para se conectar. Por padrão o mesmo nome do usuário do sistema operacional que está executando o aplicativo.

`password`

A senha a ser utilizada se o servidor requerer autenticação por senha.

`connect_timeout`

Tempo máximo para aguardar pela conexão, em segundos (escrito como uma cadeia de caracteres contendo um número inteiro decimal). Zero, ou não especificado, significa aguardar indefinidamente. Não se recomenda utilizar um tempo limite inferior a 2 segundos.

`options`

Opções de linha de comando a serem passadas para o servidor.

`tty`

Ignorado (no passado especificava para onde enviar a saída de depuração do servidor).

`sslmode`

Esta opção determina se, ou com que prioridade, será negociada uma conexão SSL com o servidor. Existem quatro modos: `disable` (desabilitado) tenta apenas uma conexão SSL não criptografada; `allow` (permitido) negocia, tentando primeiro uma conexão não-SSL, depois, se não for bem-sucedido, tenta uma conexão SSL; `prefer` (preferido) (o padrão) negocia, tentando primeiro uma conexão SSL, depois, se não for bem-sucedido, tenta uma conexão não SSL regular; `require` (requerido) tenta apenas uma conexão SSL.

Quando o PostgreSQL é compilado sem suporte a SSL a utilização da opção `require` causa um erro, enquanto as opções `allow` e `prefer` serão aceitas, mas a libpq não vai tentar de fato uma conexão SSL.

`requiressl`

Esta opção está em obsolescência em favor da definição de `sslmode`.

Se for definido como 1, é requerida uma conexão SSL com o servidor (equivale ao modo `require` de `sslmode`). A libpq recusa se conectar se o servidor não aceitar uma conexão SSL. Se for definido como 0 (o padrão), a libpq negocia o tipo de conexão com o servidor (equivale ao modo `prefer` de `sslmode`). O PostgreSQL é compilado com suporte a SSL.



### service

Nome do serviço <sup>1</sup> a ser utilizado para obter parâmetros adicionais. Especifica um nome de serviço no arquivo `pg_service.conf`, que reúne parâmetros de conexão adicionais. Permite que os aplicativos especifiquem somente o nome do serviço, para que a manutenção dos parâmetros de conexão seja feita de forma centralizada. Para obter informações sobre como configurar este arquivo deve ser visto o arquivo `share/pg_service.conf.sample` no diretório de instalação.

Se algum dos parâmetros não for especificado, então será verificada a variável de ambiente correspondente (consulte a Seção 27.11). Se a variável de ambiente também não estiver especificada, então são utilizados os padrões nativos.

### PQsetdbLogin

Estabelece uma nova conexão com o servidor de banco de dados.

```
PGconn *PQsetdbLogin(const char *pghost,
                    const char *pgport,
                    const char *pgoptions,
                    const char *pgtty,
                    const char *dbName,
                    const char *login,
                    const char *pwd);
```

É a função antecessora de `PQconnectdb` com um número fixo de parâmetros. Possui a mesma funcionalidade, exceto que os parâmetros que faltam sempre recebem os valores padrão. Deve ser escrito `NULL`, ou uma cadeia de caracteres vazia, em qualquer um dos parâmetros fixos que vai receber o valor padrão.

### PQsetdb

Estabelece uma nova conexão com o servidor de banco de dados.

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

Esta é uma macro que chama `PQsetdbLogin` com ponteiros nulos para os parâmetros `login` e `pwd`. É fornecida para manter a compatibilidade com programas muito antigos.

### PQconnectStart

### PQconnectPoll

Estabelece uma conexão com o servidor de banco de dados de uma forma não-bloqueante.

```
PGconn *PQconnectStart(const char *conninfo);
PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

Estas duas funções são utilizadas para abrir uma conexão com o servidor de banco de dados, de uma maneira que o fluxo de execução do aplicativo não fica bloqueado devido a E/S remota enquanto esta operação é realizada. O ponto central desta abordagem é que o aguardo pelo término da operação de E/S pode ocorrer dentro do laço principal do aplicativo, em vez de dentro da função `PQconnectdb`, de forma que o aplicativo possa gerenciar esta operação em paralelo com outras atividades.

A conexão com o banco de dados é feita utilizando os parâmetros encontrados na cadeia de caracteres `conninfo`, passada para a função `PQconnectStart`. Esta cadeia de caracteres possui o mesmo formato descrito acima para `PQconnectdb`.

Nem `PQconnectStart` nem `PQconnectPoll` bloqueiam, desde que certas condições sejam respeitadas:

- Utilização dos parâmetros `hostaddr` e `host` de uma forma apropriada para garantir que não sejam realizadas procura pelo nome, nem procura pelo nome reversa. Para obter detalhes deve-se ver a documentação destes parâmetros sob `PQconnectdb` acima.
- Se for chamada a função `PQtrace`, deve-se garantir que o objeto de fluxo, para onde é feita a depuração, não bloqueie.

- Deve-se garantir que o soquete está no status apropriado antes de chamar a função `PQconnectPoll`, conforme descrito abaixo.

Para iniciar uma solicitação de conexão não bloqueante, deve-se executar `conn = PQconnectStart("connection_info_string")`. Se `conn` for nulo, então a libpq não foi capaz de alocar uma nova estrutura `PGconn`. Senão é retornado um ponteiro para `PGconn` válido (embora ainda não represente uma conexão válida com o banco de dados). Após retornar da função `PQconnectStart`, deve-se executar `status = PQstatus(conn)`. Se `status` for igual a `CONNECTION_BAD`, então a função `PQconnectStart` não foi bem-sucedida.

Se a função `PQconnectStart` for bem-sucedida, o próximo estágio é uma verificação cíclica da libpq para prosseguir com a sequência de conexão. Deve ser utilizada a função `PQsocket(conn)` para obter o descritor do soquete subjacente à conexão com o banco de dados. Ciclo: Se a função `PQconnectPoll(conn)` retornar `PGRES_POLLING_READING`, aguardar até o soquete ficar pronto para ser lido (conforme indicado por `select()`, `poll()`, ou uma função do sistema semelhante). Em seguida chamar `PQconnectPoll(conn)` novamente. Caso contrário, se `PQconnectPoll(conn)` retornar `PGRES_POLLING_WRITING`, aguardar até o soquete ficar pronto para escrita, em seguida chamar `PQconnectPoll(conn)` novamente. Caso ainda seja necessário chamar `PQconnectPoll`, isto é, logo após chamar `PQconnectStart`, o comportamento deve ser o mesmo como se esta função tivesse retornado da última vez `PGRES_POLLING_WRITING`. Este ciclo deve continuar até que `PQconnectPoll(conn)` retorne `PGRES_POLLING_FAILED`, indicando que o procedimento de conexão falhou, ou `PGRES_POLLING_OK`, indicando que a conexão foi bem-sucedida.

O status da conexão pode ser verificado a qualquer momento chamando `PQstatus`. Se retornar `CONNECTION_BAD`, então o procedimento de conexão falhou; se retornar `CONNECTION_OK`, então a conexão está pronta. Estes dois status podem ser detectados igualmente a partir do valor retornado por `PQconnectPoll`, descrita acima. Também podem ocorrer outros status durante (e apenas durante) o procedimento de conexão assíncrona. Estes status indicam o estágio corrente do procedimento de conexão, podendo ser útil para fornecer informações para o usuário, por exemplo. Estes status são:

`CONNECTION_STARTED`

Aguardando a conexão ser estabelecida.

`CONNECTION_MADE`

Conexão OK; aguardando para enviar.

`CONNECTION_AWAITING_RESPONSE`

Aguardando resposta do servidor.

`CONNECTION_AUTH_OK`

Autenticação recebida; aguardando a inicialização do servidor terminar.

`CONNECTION_SSL_STARTUP`

Negociando criptografia SSL.

`CONNECTION_SETENV`

Negociando definições de parâmetro dirigida pelo ambiente.

Deve ser observado que embora estas constantes continuarão existindo (para manter a compatibilidade), um aplicativo nunca deve depender de suas ocorrências em uma determinada ordem, ou mesmo que aconteçam, ou que o status sempre seja um destes valores documentados. O aplicativo deve proceder de forma parecida com a mostrada abaixo:

```
switch(PQstatus(conn))
{
    case CONNECTION_STARTED:
        status = "Conectando...";
        break;

    case CONNECTION_MADE:
        status = "Conectado ao servidor...";
        break;
```

```

.
.
.
    default:
        status = "Conectando...";
}

```

O parâmetro de conexão `connect_timeout` é ignorado quando se utiliza `PQconnectPoll`; o aplicativo é responsável por decidir quando decorreu uma quantidade excessiva de tempo. Caso contrário, a função `PQconnectStart` seguida por `PQconnectPoll` seria equivalente à função `PQconnectdb`.

Deve ser observado que quando a função `PQconnectStart` retorna um ponteiro não nulo, deve-se chamar a função `PQfinish` ao se terminar de utilizá-la, para que seja liberada a estrutura e todos os blocos de memória associados. Isto deve ser feito mesmo que a tentativa de conexão falhe ou seja abandonada.

#### `PQconnndefaults`

Retorna as opções de conexão padrão.

```
PGconninfoOption *PQconnndefaults(void);
```

```
typedef struct
{
    char    *keyword;    /* A palavra chave da opção */
    char    *envvar;     /* O nome da variável de ambiente alternativa de falha */
    char    *compiled;   /* O nome do valor padrão compilado alternativo de falha */
    char    *val;        /* Valor corrente da opção, ou NULL */
    char    *label;      /* Rótulo para o campo no diálogo de conexão */
    char    *dispchar;   /* Caractere a ser mostrado para este campo
                        no diálogo de conexão. Os valores são:
                        ""      Mostrar o valor entrado como ele é
                        "*"     Campo de senha - esconder o valor
                        "D"     Opção de depuração - não mostrar por padrão */
    int     dispsize;    /* Tamanho do campo em caracteres para o diálogo */
} PGconninfoOption;
```

Retorna uma matriz de opções de conexão. Pode ser utilizado para determinar todas as opções possíveis de `PQconnectdb` e seus valores padrão corrente. O valor retornado aponta para uma matriz de estruturas `PGconninfoOption`, que termina por uma entrada possuindo um ponteiro nulo para `keyword`. Deve ser observado que os valores padrão corrente (campos `val`) dependem das variáveis de ambiente e outro contexto. A chamada deve tratar os dados das opções de conexão como somente para leitura.

Após a matriz de opções ser processada, esta deve ser liberada passando-a para `PQconninfoFree`. Caso não seja feito, haverá um pequeno vazamento de memória (`memory leak`) em cada chamada à função `PQconnndefaults`.

#### `PQfinish`

Fecha a conexão com o servidor. Também libera a memória utilizada pelo objeto `PGconn`.

```
void PQfinish(PGconn *conn);
```

Deve ser observado que mesmo quando a tentativa de conexão com o servidor falha (conforme indicado por `PQstatus`), o aplicativo deve chamar a função `PQfinish` para liberar a memória utilizada pelo objeto `PGconn`. O ponteiro para `PGconn` não deve ser utilizado novamente após a função `PQfinish` ter sido chamada.

#### `PQreset`

Reinicia o canal de comunicação com o servidor.

```
void PQreset(PGconn *conn);
```

Esta função fecha a conexão com o servidor e tenta restabelecer uma nova conexão com o mesmo servidor, usando exatamente os mesmos parâmetros utilizados anteriormente. Pode ser útil para recuperação de erro quando a conexão de trabalho é perdida.

PQresetStart  
PQresetPoll

Reinicia o canal de comunicação com o servidor, de uma maneira não bloqueante.

```
int PQresetStart(PGconn *conn);
```

```
PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

Estas funções fecham a conexão com o servidor e tentam restabelecer uma nova conexão com o mesmo servidor, usando exatamente os mesmos parâmetros utilizados anteriormente. Pode ser útil para recuperação de erro quando a conexão de trabalho é perdida. Diferem da função `PQreset` (acima) por atuarem de uma maneira não bloqueante. Estas funções sofrem as mesmas restrições das funções `PQconnectStart` e `PQconnectPoll`.

Para começar o reinício de conexão deve ser chamada a função `PQresetStart`. Se retornar 0, o reinício falhou. Se retornar 1 verificar ciclicamente o reinício utilizando a função `PQresetPoll` exatamente da mesma maneira como seria criada a conexão utilizando `PQconnectPoll`.

## 27.2. Funções de status da conexão

Estas funções podem ser utilizadas para indagar o status de um objeto de conexão com banco de dados existente.

**Dica:** Os programadores de aplicativos libpq devem tomar o cuidado de manter a abstração da `PGconn`. Devem ser utilizadas as funções de acesso descritas abaixo para obter o conteúdo de `PGconn`. Deve-se evitar a referência direta aos campos da estrutura `PGconn`, porque estão sujeitos a mudanças futuras (A partir da versão 6.4 do PostgreSQL, a definição da `struct` por trás de `PGconn` não é fornecida nem mesmo em `libpq-fe.h`. Caso exista um código antigo acessando diretamente os campos de `PGconn`, pode-se continuar usando esta forma incluindo também `libpq-int.h`, mas encoraja-se que o código seja corrigido em breve).

As funções abaixo retornam valores dos parâmetros estabelecidos durante a conexão. Estes valores são fixos durante a vida do objeto `PGconn`.

PQdb

Retorna o nome do banco de dados da conexão.

```
char *PQdb(const PGconn *conn);
```

PQuser

Retorna o nome do usuário da conexão.

```
char *PQuser(const PGconn *conn);
```

PQpass

Retorna a senha da conexão.

```
char *PQpass(const PGconn *conn);
```

PQhost

Retorna o nome do hospedeiro servidor da conexão.

```
char *PQhost(const PGconn *conn);
```

PQport

Retorna a porta da conexão.

```
char *PQport(const PGconn *conn);
```

PQtty

Retorna o TTY de depuração da conexão (Está obsoleto, uma vez que o servidor não presta mais atenção na definição de TTY, mas a função permanece para manter a compatibilidade com as versões anteriores).

```
char *PQtty(const PGconn *conn);
```

PQoptions

Retorna as opções de linha de comando passadas no pedido de conexão.

```
char *PQoptions(const PGconn *conn);
```

As funções abaixo retornam informações de status que podem mudar devido a operações executadas no objeto PGconn.

#### PQstatus

Retorna o status da conexão.

```
ConnStatusType PQstatus(const PGconn *conn);
```

O status pode ser um entre um conjunto de valores. Entretanto, somente dois destes valores são vistos fora de um procedimento de conexão assíncrono: CONNECTION\_OK e CONNECTION\_BAD. Uma conexão bem-sucedida com o banco de dados possui o status CONNECTION\_OK. Uma tentativa de conexão com o banco de dados mal-sucedida possui o status CONNECTION\_BAD. Normalmente, um status OK permanece assim até PQfinish, mas uma falha na conexão pode resultar em uma mudança prematura para o status CONNECTION\_BAD. Neste caso, o aplicativo pode tentar a recuperação chamando PQreset.

Com relação aos outros códigos de status que podem ser encontrados, devem ser vistas as descrições de PQconnectStart e PQconnectPoll.

#### PQtransactionStatus

Retorna o status corrente da transação no servidor.

```
PGTransactionStatusType PQtransactionStatus(const PGconn *conn);
```

O status pode ser PQTRANS\_IDLE (ocioso no momento), PQTRANS\_ACTIVE (o comando está sendo executado), PQTRANS\_INTRANS (ocioso, em um bloco de transação válido), ou PQTRANS\_INERROR (ocioso, em um bloco de transação que falhou). Se a conexão estiver ruim é relatado PQTRANS\_UNKNOWN. Somente é relatado PQTRANS\_ACTIVE quando tiver sido enviado para o servidor um comando que ainda não está completo.

### Cuidado

A função PQtransactionStatus retorna resultados incorretos quando é utilizada em um servidor PostgreSQL 7.3 com o parâmetro autocommit definido como desabilitado. A funcionalidade de auto-efetivação do lado servidor entrou em obsolescência, não existindo mais em versões posteriores do servidor.

#### PQparameterStatus

Procura a definição corrente do parâmetro no servidor.

```
const char *PQparameterStatus(const PGconn *conn, const char *paramName);
```

Certos valores de parâmetros são relatados pelo servidor automaticamente no início da conexão, ou sempre que seus valores mudam. A função PQparameterStatus pode ser utilizada para indagar estas definições. Retorna o valor corrente do parâmetro caso este seja conhecido, ou NULL se o parâmetro não for conhecido.

Os parâmetros relatados na versão corrente incluem server\_version, server\_encoding, client\_encoding, is\_superuser, session\_authorization, DateStyle, TimeZone e integer\_datetimes (server\_encoding, TimeZone e integer\_datetimes não eram relatados nas versões anteriores a 8.0). Deve ser observado que server\_version, server\_encoding e integer\_datetimes não podem mudar após a inicialização.

Os servidores com protocolo pré-3.0 não relatam as definições dos parâmetros, mas, de qualquer forma, a libpq inclui lógica para obter os valores de server\_version e client\_encoding. Encoraja-se que os aplicativos utilizem a função PQparameterStatus, em vez de código *ad hoc*<sup>2</sup> para determinar estes valores (Entretanto, deve-se estar ciente que, em uma conexão pré-3.0, quando se muda client\_encoding através de SET após o início da conexão, isto não é refletido pela função PQparameterStatus). Para server\_version deve ser vista também a função PQserverVersion, que retorna a informação de uma forma numérica muito mais fácil para fazer a comparação.

Embora o ponteiro retornado seja declarado como const, na verdade aponta para um armazenamento mutável associado à estrutura PGconn. Não é prudente assumir que o ponteiro permaneça válido entre os comandos.

#### PQprotocolVersion

Indaga o protocolo cliente/servidor sendo utilizado.

```
int PQprotocolVersion(const PGconn *conn);
```

Os aplicativos podem fazer uso desta função para determinar se certas funcionalidades são suportadas. Atualmente os valores possíveis são 2 (protocolo 2.0), 3 (protocolo 3.0), ou zero (conexão mal-sucedida). Isto não muda após a conexão ser completada, mas teoricamente pode mudar durante um reinício de conexão. Normalmente, é utilizado o protocolo 3.0 ao ser feita a comunicação com servidores PostgreSQL 7.4 ou posteriores; os servidores pré-7.4 suportam apenas o protocolo 2.0 (O protocolo 1.0 está obsoleto não sendo suportado pela libpq).

#### PQserverVersion

Retorna um inteiro representando a versão do servidor.

```
int PQserverVersion(const PGconn *conn);
```

Os aplicativos podem utilizar esta função para determinar a versão do servidor de banco de dados ao qual estão conectados. O número é formado convertendo o número principal, secundário e de revisão (*major*, *minor* e *revision*) da versão em números decimais de dois dígitos, e juntando estes números. Por exemplo, a versão 7.4.2 é retornada como 70402, e a versão 8.1 será retornada como 80100 (os zeros à esquerda não são mostrados). Retorna zero se a conexão estiver ruim.

#### PQerrorMessage

Retorna a mensagem de erro mais recente ocasionada por uma operação nesta conexão.

```
char *PQerrorMessage(const PGconn *conn);
```

Quase todas as funções da libpq definem mensagem para a função `PQerrorMessage` quando falham. Deve ser observado que pela convenção da libpq, um resultado da função `PQerrorMessage` que não esteja vazio inclui uma nova-linha no final. Quem chama não deve liberar o resultado diretamente. O resultado será liberado quando o tratador de `PGconn` associado for passado para a função `PQfinish`. Não deve ser esperado que a cadeia de caracteres do resultado permaneça a mesma entre operações na estrutura `PGconn`.

#### PQsocket

Obtém o número descritor do arquivo do soquete de conexão com o servidor. Um descritor válido será maior ou igual a 0; um resultado com valor -1 indica que no momento não existe conexão aberta com o servidor (Isto não muda durante a operação normal, mas pode mudar durante a definição ou reinício da conexão).

```
int PQsocket(const PGconn *conn);
```

#### PQbackendPID

Retorna o ID do processo (PID) do servidor que trata a conexão.

```
int PQbackendPID(const PGconn *conn);
```

O PID do servidor é útil para fins de depuração e para comparação com as mensagens do `NOTIFY` (que incluem o PID do processo servidor que está fazendo a notificação). Deve ser observado que o PID pertence ao processo que executa no hospedeiro servidor de banco de dados, e não no hospedeiro local!

#### PQgetssl

Retorna a estrutura SSL utilizada na conexão, ou nulo se não estiver sendo utilizada a SSL.

```
SSL *PQgetssl(const PGconn *conn);
```

Esta estrutura pode ser utilizada para verificar os níveis de criptografia, verificar os certificados do servidor, e outras informações. Para obter informações sobre esta estrutura, deve ser consultada a documentação do OpenSSL.

Para ser obtido o protótipo correto desta função, deve ser definido `USE_SSL`. Quando isto é feito, também é incluído automaticamente `ssl.h` do OpenSSL.

## 27.3. Funções de execução de comando

Uma vez que a conexão com o servidor de banco de dados tenha sido estabelecida com sucesso, são usadas as funções descritas abaixo para executar comandos e consultas SQL.

### 27.3.1. Funções principais

#### PQexec

Submete um comando ao servidor e aguarda pelo resultado.

```
PGresult *PQexec(PGconn *conn, const char *command);
```

Retorna um ponteiro para `PGresult`, ou um ponteiro nulo. Geralmente é retornado um ponteiro não nulo, exceto nas condições de falta de memória ou erros graves, como a impossibilidade de enviar o comando para o servidor. Caso seja retornado um ponteiro nulo, este deve ser tratado como um resultado `PGRES_FATAL_ERROR`. Deve ser utilizada a função `PQerrorMessage` para obter informações adicionais sobre erros deste tipo.

É permitido incluir vários comandos SQL (separados por ponto-e-vírgula) na cadeia de caracteres do comando. Quando são enviados vários comandos na mesma chamada à `PQexec`, estes são processados em uma única transação, a menos que existam comandos `BEGIN` e `COMMIT` explícitos incluídos na cadeia de caracteres enviada, para dividi-la em várias transações. Entretanto, deve ser observado que a estrutura `PGresult` retornada descreve apenas o resultado do último comando da cadeia de caracteres executado. Se um dos comandos falhar, o processamento da cadeia de caracteres é interrompido neste comando, e a estrutura `PGresult` retornada descreve a condição de erro.

#### `PQexecParams`

Submete um comando ao servidor e aguarda pelo resultado, com a capacidade de passar parâmetros separadamente do texto do comando SQL.

```
PGresult *PQexecParams(PGconn *conn,
                        const char *command,
                        int nParams,
                        const Oid *paramTypes,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

A função `PQexecParams` é semelhante à função `PQexec`, mas oferece uma funcionalidade adicional: os valores dos parâmetros podem ser especificados separadamente da cadeia de caracteres do comando, e pode ser solicitado que o resultado do comando retorne no modo texto ou binário. A função `PQexecParams` é suportada apenas pelas conexões que utilizam o protocolo 3.0 ou mais recente; falha quando é utilizado o protocolo 2.0.

Quando são utilizados parâmetros, estes são referenciados na cadeia de caracteres do comando como `$1`, `$2`, etc. `nParams` é o número de parâmetros fornecidos; é o comprimento das matrizes `paramTypes[]`, `paramValues[]`, `paramLengths[]` e `paramFormats[]` (Os ponteiros para as matrizes podem ser `NULL` quando `nParams` for zero). `paramTypes[]` especifica, por OID, os tipos de dado a serem atribuídos aos símbolos dos parâmetros. Se `paramTypes` for `NULL`, ou algum determinado elemento da matriz for zero, o servidor atribui o tipo de dado para o símbolo do parâmetro da mesma maneira que faria para um literal cadeia de caracteres sem tipo. `paramValues[]` especifica os valores dos parâmetros. Nesta matriz um ponteiro nulo significa que o parâmetro correspondente é nulo; senão o ponteiro aponta para uma cadeia de caracteres de texto terminada por zero (para o formato texto), ou dados binários no formato esperado pelo servidor (para formato binário). `paramLengths[]` especifica o comprimento dos dados dos parâmetros com formato binário. É ignorado para parâmetros nulos e parâmetros com formato texto. O ponteiro para a matriz pode ser nulo quando não há parâmetros binários. `paramFormats[]` especifica se os parâmetros são do tipo texto (colocando-se o valor zero na matriz), ou binário (colocando-se o valor um na matriz). Se o ponteiro para a matriz for nulo, presume-se que todos os parâmetros sejam do tipo texto. `resultFormat` é zero para os resultados serem recebidos no formato texto, ou um para os resultados serem recebidos no formato binário (Atualmente não há como receber colunas de resultado diferentes com formatos diferentes, embora isto seja possível no protocolo subjacente).

A principal vantagem da função `PQexecParams` sobre a função `PQexec` é que os valores dos parâmetros podem ficar separados da cadeia de caracteres de comando, evitando a necessidade de colocar escapes e apóstrofes, que é entediante e sujeito a erros. Diferentemente de `PQexec`, a função `PQexecParams` permite no máximo um comando SQL em uma cadeia de caracteres (Pode existir mais de um ponto-e-vírgula na cadeia de caracteres, mas não mais de um comando não vazio). Esta é uma limitação do protocolo subjacente, mas possui alguma utilidade como defesa adicional contra ataques de injeção-SQL.

#### `PQprepare`

Submete uma solicitação para criar uma declaração preparada com os parâmetros fornecidos, e aguarda completar.

```
PGresult *PQprepare(PGconn *conn,
                    const char *stmtName,
                    const char *query,
```

```
int nParams,
const Oid *paramTypes);
```

A função `PQprepare` cria uma declaração preparada para execução posterior através de `PQexecPrepared`. Esta funcionalidade permite que comandos a serem utilizados repetidas vezes sejam analisados e planejados somente uma vez, em vez de cada vez que são utilizados. A função `PQprepare` é suportada apenas pelas conexões que utilizam o protocolo 3.0 ou mais recente; falha quando é utilizado o protocolo 2.0.

Esta função cria uma declaração preparada chamada `stmtName` a partir da cadeia de caracteres `query`, que deve conter um único comando SQL. `stmtName` pode ser igual a "" para ser criada uma declaração sem nome, caso em que uma declaração sem nome pré-existente é substituída automaticamente; de outra forma, ocasiona erro se o nome da declaração já estiver definida na sessão corrente. Se for usado algum parâmetro, estes devem ser referenciados no comando como \$1, \$2, etc. `nParams` é o número de parâmetros para os quais os tipos são previamente especificados na matriz `paramTypes[]` (O ponteiro para a matriz pode ser nulo quando `NULL` quando `nParams` for zero). `paramTypes[]` especifica, por OID, os tipos de dado a serem atribuídos aos símbolos dos parâmetros. Se `paramTypes` for `NULL`, ou algum determinado elemento da matriz for zero, o servidor atribui o tipo de dado para o símbolo do parâmetro da mesma maneira que faria para um literal cadeia de caracteres sem tipo. Além disso, o comando pode utilizar símbolos de parâmetro com números maiores que `nParams`; os tipos de dado para estes símbolos também serão inferidos.

Da mesma forma que na função `PQexec`, o resultado normalmente é um objeto `PGresult` cujo conteúdo indica sucesso ou falha do lado servidor. Um resultado nulo indica falta de memória ou incapacidade enviar o comando. Deve ser utilizada a função `PQerrorMessage` para obter mais informações sobre erros deste tipo.

No presente momento não há nenhuma forma de determinar o verdadeiro tipo de dado inferido para qualquer parâmetro cujo tipo não foi especificado através de `paramTypes[]`. Esta é uma omissão da libpq que, provavelmente, será corrigida em uma versão futura.

As declarações preparadas a serem utilizadas pela função `PQexecPrepared` também podem ser criadas utilizando declarações `PREPARE` do SQL (Mas a função `PQprepare` é mais flexível, uma vez que não requer que os tipos dos parâmetros sejam previamente especificados). Além disso, embora não haja nenhuma função na libpq para remover uma declaração preparada, a declaração `DEALLOCATE` do SQL pode ser utilizada para esta finalidade.

#### `PQexecPrepared`

Envia uma solicitação para executar uma declaração preparada com determinados parâmetros, e aguarda pelo resultado.

```
PGresult *PQexecPrepared(PGconn *conn,
                          const char *stmtName,
                          int nParams,
                          const char * const *paramValues,
                          const int *paramLengths,
                          const int *paramFormats,
                          int resultFormat);
```

A função `PQexecPrepared` é semelhante à função `PQexecParams`, mas o comando a ser executado é especificado fornecendo-se o nome de uma declaração previamente preparada, em vez de fornecer a cadeia de caracteres do comando. Esta funcionalidade permite que comandos a serem utilizados repetidas vezes sejam analisados e planejados somente uma vez, em vez de cada vez que são utilizados. A declaração deve ter sido preparada anteriormente na sessão corrente. A função `PQprepare` é suportada apenas pelas conexões que utilizam o protocolo 3.0 ou mais recente; falha quando é utilizado o protocolo 2.0.

Os parâmetros são idênticos aos da função `PQexecParams`, exceto que é fornecido o nome da declaração preparada em vez da cadeia de caracteres do comando, e o parâmetro `paramTypes[]` não está presente (não é necessário, uma vez que os tipos dos parâmetros da declaração preparada foram determinados quando esta foi criada).

A estrutura `PGresult` encapsula o resultado retornado pelo servidor. Os programadores de aplicativos libpq devem tomar o cuidado de manter a abstração de `PGresult`. Devem ser utilizadas as funções de acesso abaixo para obter o conteúdo de `PGresult`. Deve ser evitado a referência direta aos campos da estrutura `PGresult`, porque estes estão sujeitos a modificações futuras.



**PQresultStatus**

Retorna o status do resultado do comando.

```
ExecStatusType PQresultStatus(const PGresult *res);
```

A função `PQresultStatus` pode retornar um dos seguintes valores:

`PGRES_EMPTY_QUERY`

A cadeia de caracteres enviada ao servidor estava vazia.

`PGRES_COMMAND_OK`

Término bem-sucedido de um comando que não retorna dados.

`PGRES_TUPLES_OK`

Término bem-sucedido de um comando que retorna dados (como `SELECT` ou `SHOW`).

`PGRES_COPY_OUT`

Transferência de dados para fora do servidor iniciada (`Copy Out`).

`PGRES_COPY_IN`

Transferência de dados para dentro do servidor iniciada (`Copy In`).

`PGRES_BAD_RESPONSE`

A resposta do servidor não foi compreendida.

`PGRES_NONFATAL_ERROR`

Ocorreu um erro não fatal (nota ou advertência).

`PGRES_FATAL_ERROR`

Ocorreu um erro fatal.

Se o status do resultado for `PGRES_TUPLES_OK`, então podem ser utilizadas as funções descritas abaixo para obter as linhas retornadas pela consulta. Deve ser observado que um comando `SELECT` que não retorna nenhuma linha ainda assim mostra `PGRES_TUPLES_OK`. `PGRES_COMMAND_OK` é utilizado em comandos que nunca retornam linhas (`INSERT`, `UPDATE`, etc.). Uma resposta do tipo `PGRES_EMPTY_QUERY` pode indicar um erro no programa cliente.

Um resultado com o status `PGRES_NONFATAL_ERROR` nunca será retornado diretamente por `PQexec` ou outra função de execução de comandos; em vez disso, os resultados deste tipo são passados para o processador de observações (consulte a Seção 27.10).

**PQresStatus**

Converte o tipo enumerado retornado pela função `PQresultStatus` em uma constante cadeia de caracteres que descreve o código do status. Quem chama não deve liberar o resultado.

```
char *PQresStatus(ExecStatusType status);
```

**PQresultErrorMessage**

Retorna a mensagem de erro associada ao comando, ou uma cadeia de caracteres vazia caso não tenha havido erro.

```
char *PQresultErrorMessage(const PGresult *res);
```

Caso tenha havido um erro, a cadeia de caracteres retornada inclui um caractere de nova-linha no final. Quem chama não deve liberar o resultado diretamente. O resultado será liberado quando o tratador `PGresult` associado for passado para a função `PQclear`.

Uma chamada a `PQerrorMessage` imediatamente após a chamada a `PQexec` ou `PQgetResult` (na mesma conexão), retorna a mesma cadeia de caracteres que `PQresultErrorMessage` (no resultado). Entretanto, a estrutura `PGresult` retém a mensagem de erro até ser destruída, enquanto a mensagem de erro da conexão muda quando as operações seguintes são realizadas. Deve ser utilizada a função `PQresultErrorMessage` quando se deseja conhecer o status associado a uma determinada estrutura `PGresult`; deve ser utilizada a função `PQerrorMessage` quando se deseja conhecer o status da última operação na conexão.

`PQresultErrorField`

Retorna um campo individual de um relato de erro.

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

O parâmetro `fieldcode` é um identificador de campo de erro; devem ser vistos os símbolos listados abaixo. Retorna `NULL` quando `PGresult` não é um resultado de erro ou de advertência, ou não inclui o campo especificado. Os valores dos campos normalmente não incluem o caractere de nova-linha no final. Quem chama não deve liberar o resultado diretamente. O resultado será liberado quando o tratador da estrutura `PGresult` associado for passado para a função `PQclear`.

Estão disponíveis os seguintes códigos de campo:

`PG_DIAG_SEVERITY`

A severidade; o conteúdo do campo pode ser `ERROR`, `FATAL` ou `PANIC` (em uma mensagem de erro), ou `WARNING`, `NOTICE`, `DEBUG`, `INFO` ou `LOG` (em uma mensagem de observação), ou uma tradução localizada de um destes valores. Sempre presente.

`PG_DIAG_SQLSTATE`

O código `SQLSTATE` do erro, que identifica o tipo de erro ocorrido; pode ser utilizado pelos aplicativos clientes para realizar operações específicas (como o tratamento de erros) em resposta a um erro do banco de dados. Para obter a lista dos códigos `SQLSTATE` possíveis, deve ser visto o Apêndice A. Este campo não muda com o idioma, e está sempre presente.

`PG_DIAG_MESSAGE_PRIMARY`

A principal mensagem de erro humanamente legível (tipicamente uma linha). Sempre presente.

`PG_DIAG_MESSAGE_DETAIL`

Detalhe: uma mensagem de erro secundário opcional contendo mais detalhes sobre o problema. Pode ter várias linhas.

`PG_DIAG_MESSAGE_HINT`

Dica: uma sugestão opcional sobre o que fazer para resolver o problema. Tem por intenção diferir do detalhe por oferecer conselho (potencialmente não apropriado), em vez de simples fatos. Pode ter várias linhas.

`PG_DIAG_STATEMENT_POSITION`

Uma cadeia de caracteres contendo um inteiro decimal que indica a posição do cursor de erro, como um índice dentro da cadeia de caracteres original da declaração. O primeiro caractere possui o índice 1, e as posições são medidas em caracteres e não em bytes.

`PG_DIAG_INTERNAL_POSITION`

É definido da mesma maneira que o campo `PG_DIAG_STATEMENT_POSITION`, mas é utilizado quando a posição do cursor se refere a um comando gerado internamente, em vez de um comando submetido pelo cliente. O campo `PG_DIAG_INTERNAL_QUERY` sempre está presente quando este campo está presente.

`PG_DIAG_INTERNAL_QUERY`

O texto do comando gerado internamente que falhou. Pode ser, por exemplo, um comando SQL emitido por uma função PL/pgSQL.

`PG_DIAG_CONTEXT`

Uma indicação do contexto em que o erro ocorreu. Atualmente inclui o histórico da pilha de chamada (`call stack traceback`) das funções da linguagem procedural e dos comandos gerados internamente. O histórico contém uma entrada por linha, com a mais recente na frente.

`PG_DIAG_SOURCE_FILE`

O nome do arquivo do local do código fonte onde o erro foi relatado.

`PG_DIAG_SOURCE_LINE`

O número da linha do local do código fonte onde o erro foi relatado.

**PG\_DIAG\_SOURCE\_FUNCTION**

O nome da função do código fonte que relatou o erro.

O cliente é responsável pela formatação da informação mostrada, conforme suas necessidades; em particular, as linhas longas devem ser quebradas conforme seja necessário. Os caracteres de nova-linha que aparecem nos campos de mensagem de erro devem ser tratados como quebra de parágrafo, e não como quebra de linha.

As mensagens de erro geradas internamente pela libpq possuem mensagem primária e severidade, mas tipicamente nenhum outro campo. Os erros retornados por um servidor com protocolo pré-3.0 incluem mensagem primária e severidade, e algumas vezes mensagem de detalhe, mas nenhum outro campo.

Deve ser observado que os campos de erro estão disponíveis apenas nos objetos `PGresult`, e não nos objetos `PGconn`; não existe nenhuma função chamada `PQerrorMessage`.

**PQclear**

Libera o armazenamento associado a `PGresult`. Todo resultado de comando deve ser liberado através de `PQclear` quando não for mais necessário.

```
void PQclear(PGresult *res);
```

O objeto `PGresult` pode ser mantido pelo tempo que for necessário; não precisa ir embora quando se submete um novo comando, nem mesmo quando se fecha a conexão. Para liberá-lo, deve ser chamada a função `PQclear`. Caso não seja feito acarreta perda de memória pelo aplicativo.

**PQmakeEmptyPGresult**

Constrói um objeto `PGresult` vazio com o status fornecido.

```
PGresult* PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

Esta é uma função interna da libpq para alocar e inicializar um objeto `PGresult` vazio. É exportado porque alguns aplicativos encontram utilidade na geração de objetos de resultado (em particular objetos com status de erro). Se `conn` não for nulo, e `status` indica um erro, a mensagem de erro corrente da conexão especificada é copiada para `PGresult`. Deve ser observado que no final a função `PQclear` deverá ser chamada para o objeto, da mesma forma que com a estrutura `PGresult` retornada pela própria libpq.

**27.3.2. Obtenção da informação de resultado do comando**

Estas funções são utilizadas para extrair informações do objeto `PGresult` que representa o resultado de um comando bem-sucedido (ou seja, um comando com o status `PGRES_TUPLES_OK`). Para os objetos com outros valores de status, estas funções agem como se o resultado tivesse zero linhas e zero colunas.

**PQntuples**

Retorna o número de linhas (tuplas) presentes no resultado do comando.

```
int PQntuples(const PGresult *res);
```

**PQnfields**

Retorna o número de colunas (campos) de cada linha presente no resultado do comando.

```
int PQnfields(const PGresult *res);
```

**PQfname**

Retorna o nome da coluna associado a um determinado número de coluna. Os números das colunas começam por zero. Quem chama não deve liberar o resultado diretamente. O resultado é liberado quando o tratador de `PGresult` associado é passado para a função `PQclear`.

```
char *PQfname(const PGresult *res,
               int column_number);
```

Retorna `NULL` quando o número da coluna está fora do intervalo.

**PQfnumber**

Retorna o número da coluna associado a um determinado nome de coluna.

```
int PQfnumber(const PGresult *res,
```

```
const char *column_name);
```

Retorna -1 se o nome fornecido não corresponder a nenhuma coluna.

O nome fornecido é tratado da mesma maneira que um identificador de um comando SQL, ou seja, as letras são convertidas em minúsculas a menos que estejam entre aspas. Por exemplo, dada a saída da consulta gerada pelo comando SQL

```
select 1 as FOO, 2 as "BAR";
```

seriam obtidos os seguintes resultados:

```
PQfname(res, 0)           foo
PQfname(res, 1)           BAR
PQfnumber(res, "FOO")      0
PQfnumber(res, "foo")      0
PQfnumber(res, "BAR")     -1
PQfnumber(res, "\"BAR\"")  1
```

#### PQftable

Retorna o OID da tabela da qual a coluna especificada foi trazida. Os números das colunas começam por 0.

```
Oid PQftable(const PGresult *res,
             int column_number);
```

Retorna InvalidOid se o número da coluna estiver fora do intervalo, ou se a coluna especificada não for uma referência simples a uma coluna de tabela, ou quando é utilizado um protocolo anterior ao 3.0. Pode ser consultada a tabela `pg_class` para determinar exatamente qual tabela é referenciada.

O tipo `Oid` e a constante `InvalidOid` são definidas quando é incluído o arquivo de cabeçalho da libpq. Ambos serão algum tipo inteiro.

#### PQftablecol

Retorna o número da coluna (dentro de sua tabela), da coluna que produz a coluna de resultado da consulta especificada. Os números de coluna dos resultados das consultas começam por 0, mas as colunas das tabela possuem números diferentes de zero.

```
int PQftablecol(const PGresult *res,
                int column_number);
```

Retorna zero se o número da coluna estiver fora do intervalo, ou se a coluna especificada não for uma referência simples a uma coluna de tabela, ou quando é utilizado um protocolo anterior ao 3.0.

#### PQfformat

Retorna o código do formato que indica o formato de uma determinada coluna. Os números das colunas começam por 0.

```
int PQfformat(const PGresult *res,
              int column_number);
```

Código de forma zero indica representação textual dos dados, enquanto o código de formato um indica representação binária (os outros códigos estão reservados para definições futuras).

#### PQftype

Retorna o tipo de dado associado com o número de coluna especificado. O inteiro retornado é o número de OID interno do tipo. Os números de coluna começam por 0.

```
Oid PQftype(const PGresult *res,
            int column_number);
```

Pode ser consultada a tabela do sistema `pg_type` para obter os nomes e propriedades de vários tipos de dado. Os OIDs dos tipos de dado nativos estão definidos no arquivo `src/include/catalog/pg_type.h` na árvore do código fonte.

#### PQfmod

Retorna o modificador de tipo da coluna associada com o número de coluna especificado. Os números de coluna começam por 0.

```
int PQfmod(const PGresult *res,
           int column_number);
```

A interpretação dos valores de modificador é específica do tipo; tipicamente indicam os limites de precisão ou de tamanho. O valor -1 é utilizado para indicar “nenhuma informação disponível”. A maior parte dos tipos de dado não utilizam modificadores, e neste caso o valor é sempre -1.

#### PQfsize

Retorna o tamanho, em bytes, da coluna associada ao número de coluna especificado. Os números de coluna começam por 0.

```
int PQfsize(const PGresult *res,
            int column_number);
```

A função `PQfsize` retorna o espaço alocado para esta coluna na linha do banco de dados ou, em outras palavras, o tamanho da representação interna do servidor do tipo de dado (Na verdade, não é muito útil para os clientes). Um valor negativo indica que o tipo de dado é de tamanho variável.

#### PQbinaryTuples

Retorna 1 se a estrutura `PGresult` contém dados binários, e 0 se contém dados na forma de texto.

```
int PQbinaryTuples(const PGresult *res);
```

Esta função está em obsolescência (exceto para uso na conexão com `COPY`), porque é possível que uma única estrutura `PGresult` contenha dados na forma de texto para algumas colunas, e dados binários em outras colunas. É preferida a função `PQfformat`. A função `PQbinaryTuples` retorna 1 somente se todas as colunas do resultado forem binárias (formato 1).

#### PQgetvalue

Retorna um único valor de campo de uma linha de `PGresult`. Os números das linhas e das colunas começam por 0. Quem chama não deve liberar o resultado diretamente. O resultado será liberado quando o tratador de `PGresult` associado for passado para a função `PQclear`.

```
char *PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

Para os dados no formato texto, a representação dos valores retornados por `PQgetvalue` é uma cadeia de caracteres terminada por nulo do valor do campo. Para os dados no formato binário, o valor está na representação binária determinada pelas funções `typsend` e `typereceive` do tipo de dado (Na verdade, neste caso o valor é seguido por um byte zero também, mas normalmente isto não é útil, uma vez que o valor pode conter nulos incorporados).

É retornada uma cadeia de caracteres vazia se o valor do campo for nulo. Deve ser vista a função `PQgetisnull` para distinguir valores nulos de cadeias de caracteres vazias.

O ponteiro retornado por `PQgetvalue` aponta para um armazenamento que faz parte da estrutura `PGresult`. O dado apontado não deve ser modificado, e os dados devem ser explicitamente copiados para outro local de armazenamento se forem ser utilizados após o tempo de vida da própria estrutura `PGresult`.

#### PQgetisnull

Testa o campo com relação ao valor nulo. Os números de linha e de coluna começam por 0.

```
int PQgetisnull(const PGresult *res,
                int row_number,
                int column_number);
```

Esta função retorna 1 se o campo for nulo, e 0 se contiver um valor não nulo (Deve ser observado que a função `PQgetvalue` retorna uma cadeia de caracteres vazia, e não um ponteiro nulo, para um campo nulo).

#### PQgetlength

Retorna o verdadeiro comprimento do valor do campo em bytes. Os números de linha e de coluna começam por 0.

```
int PQgetlength(const PGresult *res,
                int row_number,
                int column_number);
```

Este é o verdadeiro comprimento dos dados para o valor de dado em particular, ou seja, o tamanho do objeto apontado por `PQgetvalue`. Para dados no formato texto, é o mesmo que `strlen()`. Para o formato binário, esta é uma informação essencial. Deve ser observado que *não* se deve depender da função `PQfsize` para obter o verdadeiro comprimento de dado.

#### PQprint

Imprime todas as colunas e, opcionalmente, os nomes das colunas, para um fluxo de saída especificado.

```
void PQprint(FILE *fout,          /* fluxo de saída */
             const PGresult *res,
             const PQprintOpt *po);

typedef struct {
    pqbool  header;      /* print output field headings and row count */
    pqbool  align;       /* fill align the fields */
    pqbool  standard;    /* old brain dead format */
    pqbool  html3;       /* output HTML tables */
    pqbool  expanded;    /* expand tables */
    pqbool  pager;       /* use pager for output if needed */
    char    *fieldSep;   /* field separator */
    char    *tableOpt;   /* attributes for HTML table element */
    char    *caption;    /* HTML table caption */
    char    **fieldName; /* null-terminated array of replacement field names */
} PQprintOpt;
```

Esta função era anteriormente utilizada por `psql` para imprimir os resultados das consultas, mas este não é mais o caso. Deve ser observado que esta função assume que todos os dados estão no formato texto.

### 27.3.3. Obter informações do resultado de outros comandos

Estas funções são utilizadas para extrair informações de objetos `PGresult` que não são resultados do comando `SELECT`.

#### PQcmdStatus

Retorna a marca de status do comando SQL que gerou `PGresult`.

```
char *PQcmdStatus(PGresult *res);
```

Normalmente é apenas o nome do comando, mas pode incluir dados adicionais como o número de linhas processadas. Quem chama não deve liberar o resultado diretamente. O resultado será liberado quando o tratador de `PGresult` associado for passado para a função `PQclear`.

#### PQcmdTuples

Retorna o número de linhas afetadas pelo comando SQL.

```
char *PQcmdTuples(PGresult *res);
```

Esta função retorna a cadeia de caracteres que contém o número de linhas afetadas pela declaração SQL que gerou `PGresult`. Esta função pode ser utilizada apenas após a execução dos comandos `INSERT`, `UPDATE`, `DELETE`, `MOVE` e `FETCH`, ou em um `EXECUTE` de uma declaração preparada que contenha um comando `INSERT`, `UPDATE` ou `DELETE`. Se o comando que gerou `PGresult` for alguma coisa diferente, a função `PQcmdTuples` retorna uma cadeia de caracteres vazia. Quem chama não deve liberar o valor retornado `PGresult` associado for passado para a função `PQclear`.

#### PQoidValue

Retorna o OID da linha inserida, se o comando SQL foi um `INSERT` que inseriu exatamente uma linha numa tabela que possui OIDs, ou um `EXECUTE` de uma declaração preparada contendo um comando `INSERT` apropriado. Senão, esta função retorna `InvalidOid`. Esta função também retorna `InvalidOid` se a tabela afetada pelo comando `INSERT` não contiver OIDs.

```
Oid PQoidValue(const PGresult *res);
```

#### PQoidStatus

Retorna uma cadeia de caracteres com o OID da linha inserida, se o comando SQL foi um `INSERT` que inseriu exatamente uma linha, ou um `EXECUTE` de uma declaração preparada contendo um comando `INSERT` apropriado (A

cadeia de caracteres será 0 se o comando `INSERT` não inserir exatamente uma linha, ou se a tabela de destino não possuir OIDs). Se o comando não for um `INSERT`, é retornada uma cadeia de caracteres vazia.

```
char *PQoidStatus(const PGresult *res);
```

Esta função está em obsolescência em favor da função `PQoidValue`. Não é segura quanto a vários fluxos de execução (threads).

### 27.3.4. Escapes em cadeia de caracteres incluídas em comandos SQL

A função `PQescapeString` coloca escapes em cadeia de caracteres a serem utilizadas dentro de comandos SQL; É útil quando são inseridos valores de dados como constantes literais em comandos SQL. Certos caracteres (como apóstrofes e contrabarras) devem receber escape para evitar que sejam interpretados de forma especial pelo analisador de SQL. A função `PQescapeString` realiza esta operação.

**Dica:** É de especial importância fazer o escape apropriado ao se trabalhar com cadeias de caracteres que foram recebidas de fonte não segura. Senão, existe um risco de segurança: fica-se vulnerável ao ataque de “injeção-SQL”, onde comandos SQL indesejados são introduzidos no banco de dados.

Deve ser observado que não é necessário, nem correto, fazer o escape quando os valores dos dados são passados como parâmetros em separado para a função `PQexecParams` ou suas rotinas relacionadas.

```
size_t PQescapeString (char *to, const char *from, size_t length);
```

O parâmetro `from` aponta para o primeiro caractere da cadeia de caracteres que vai receber escapes, e o parâmetro `length` especifica o número de caracteres desta cadeia de caracteres. Não é requerido um byte zero para terminar, e este não deve ser contado em `length` (Se um byte zero terminador for encontrado antes que `length` bytes sejam processados, a função `PQescapeString` pára no zero; o comportamento é, portanto, bem semelhante ao da função `strncpy`). O parâmetro `to` deve apontar para um buffer capaz de conter pelo menos um caractere a mais que o dobro do valor de `length`, senão o comportamento é indefinido. Uma chamada à função `PQescapeString` escreve uma versão com escapes da cadeia de caracteres `from` para o buffer `to`, substituindo os caracteres especiais para que estes não possam causar nenhum problema, e adicionando o byte zero terminador. Os apóstrofes que devem envolver os literais cadeia de caracteres do PostgreSQL não são incluídos na cadeia de caracteres do resultado; estes devem ser colocados no comando SQL onde o resultado é inserido.

A função `PQescapeString` retorna o número de caracteres escritos no parâmetro `to`, sem incluir o byte zero terminador.

O comportamento torna-se indefinido quando há superposição do parâmetro `to` com o parâmetro `from`.

### 27.3.5. Escapes em cadeias binárias incluídas em comandos SQL

`PQescapeBytea`

Faz escape em dados binários a serem utilizados dentro de comandos SQL com o tipo `bytea`. Assim como em `PQescapeString`, somente é utilizada ao inserir dados diretamente na cadeia de caracteres do comando SQL.

```
unsigned char *PQescapeBytea(const unsigned char *from,
                             size_t from_length,
                             size_t *to_length);
```

Certos valores de byte *devem* receber escape (mas todos os valores de byte *podem* receber escape) quando utilizados como parte de um literal `bytea` em uma declaração SQL. Em geral, para fazer o escape de um byte, este é convertido em um número octal de três dígitos igual ao valor do octeto, e precedido por duas contrabarras. Os caracteres apóstrofo (') e contrabarra (\) possuem alternativas especiais de seqüência de escape. Para obter informações adicionais deve ser consultada a Seção 8.4. A função `PQescapeBytea` realiza esta operação, fazendo escape apenas do menor número de bytes necessários.

O parâmetro `from` aponta para o primeiro byte da cadeia de caracteres que vai receber os escapes, e o parâmetro `from_length` fornece o número de bytes nesta cadeia binária (Um byte zero terminador não é necessário nem contado). O parâmetro `to_length` aponta para uma variável que conterá o comprimento da cadeia de caracteres com escapes resultante. O comprimento da cadeia de caracteres resultante inclui o byte zero terminador do resultado.

A função `PQescapeBytea` retorna uma versão com escapes da cadeia binária do parâmetro `from` na memória alocada pela função `malloc()`. Esta memória deve ser liberada através da função `PQfreemem` quando o resultado não

for mais necessário. A cadeia retornada possui todos os caracteres especiais substituídos para que possam ser adequadamente processados pelo analisador de literal cadeia de caracteres do PostgreSQL, e pela função de entrada de `bytea`. O byte zero terminador também é adicionado. Os apóstrofes que devem envolver os literais cadeia de caracteres do PostgreSQL não fazem parte da cadeia de caracteres do resultado.

#### PQunescapeBytea

Converte a representação com escapes dos dados binários em dados binários — o inverso da função `PQescapeBytea`. Isto é necessário ao receber dados `bytea` no formato texto, mas não quando for recebido no formato binário.

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t *to_length);
```

O parâmetro `from` aponta para uma cadeia de caracteres com escapes como a que pode ser retornada pela função `PQgetvalue` quando esta é aplicada a uma coluna `bytea`. A função `PQunescapeBytea` converte esta representação em cadeia de caracteres na representação binária. É retornado um ponteiro para um buffer alocado pela função `malloc()`, ou nulo se ocorrer um erro, e coloca o tamanho do buffer no parâmetro `to_length`. O resultado deve ser liberado através da função `PQfreemem` quando não for mais necessário.

#### PQfreemem

Libera a memória alocada pela `libpq`.

```
void PQfreemem(void *ptr);
```

Libera a memória alocada pela `libpq`, em particular pelas funções `PQescapeBytea`, `PQunescapeBytea`, e `PQnotifies`. É necessária pelo Microsoft Windows, que não consegue liberar memória entre DLLs, a menos que sejam utilizadas DLLs com vários fluxos de execução (`multithreaded`) (/MD no VC6). Nas outras plataformas, esta função é a mesma função da biblioteca padrão `free()`.

## 27.4. Processamento de comandos assíncronos

A função `PQexec` é adequada para a submissão de comandos em aplicativos normais, síncronos. Entretanto, possui algumas deficiências importantes para alguns usuários:

- A função `PQexec` aguarda o comando completar. O aplicativo pode ter outra tarefa para realizar (tal como manter a interface do usuário) e, neste caso, não deseja ficar bloqueado aguardando pela resposta.
- Uma vez que a execução do aplicativo cliente fica suspensa enquanto este aguarda pelo resultado, é difícil para o aplicativo cliente decidir que gostaria de cancelar o comando em andamento (Pode ser feito através de um tratador de sinais, mas não de outra forma).
- A função `PQexec` pode retornar apenas uma estrutura `PGresult`. Se a cadeia de caracteres submetida contiver vários comandos SQL, então todas as estruturas `PGresult`, menos a última, são desconsideradas pela função `PQexec`.

Os aplicativos insatisfeitos com estas limitações podem, em vez desta, utilizar as funções subjacentes a partir das quais `PQexec` é construída: `PQsendQuery` e `PQgetResult`. Também existem as funções `PQsendQueryParams`, `PQsendPrepare` e `PQsendQueryPrepared`, que podem ser utilizadas com `PQgetResult` para duplicar as funcionalidades de `PQexecParams`, `PQprepare` e `PQexecPrepared`, respectivamente.

#### PQsendQuery

Submete o comando ao servidor sem aguardar pelos resultados. Retorna 1 quando o envio do comando é bem-sucedido, e 0 caso contrário (neste caso, deve ser usada a função `PQerrorMessage` para obter informações adicionais sobre a falha).

```
int PQsendQuery(PGconn *conn, const char *command);
```

Após uma chamada bem-sucedida à função `PQsendQuery`, deve ser chamada a função `PQgetResult` uma ou mais vezes para obter os resultados. A função `PQsendQuery` não deve ser chamada novamente (na mesma conexão), até que a função `PQgetResult` retorne um ponteiro nulo, indicando que o comando completou.

#### PQsendQueryParams

Submete o comando e os parâmetros em separado para o servidor, sem aguardar pelos resultados.

```
int PQsendQueryParams(PGconn *conn,
                      const char *command,
```



```

int nParams,
const Oid *paramTypes,
const char * const *paramValues,
const int *paramLengths,
const int *paramFormats,
int resultFormat);

```

É equivalente à função `PQsendQuery`, exceto que os parâmetros podem ser especificados separados da cadeia de caracteres do comando. Os parâmetros da função são tratados de forma idêntica aos da função `PQexecParams`. Da mesma forma que a função `PQexecParams`, não funciona em conexões com protocolo 2.0, e permite apenas um comando na cadeia de caracteres de comando.

#### `PQsendPrepare`

Envia uma solicitação para criar uma declaração preparada com os parâmetros fornecidos, sem aguardar completar.

```

int PQsendPrepare(PGconn *conn,
                  const char *stmtName,
                  const char *query,
                  int nParams,
                  const Oid *paramTypes);

```

Esta é a versão assíncrona da função `PQprepare`: retorna 1 se for capaz de enviar a solicitação, e 0 caso contrário. Após uma chamada bem sucedida, deve ser chamada a função `PQgetResult` para determinar se a criação da declaração preparada no servidor foi bem-sucedida. Os parâmetros desta função são tratados de maneira idêntica aos da função `PQprepare`. Da mesma forma que a função `PQprepare`, não funciona em conexões com protocolo 2.0.

#### `PQsendQueryPrepared`

Envia solicitação para executar a declaração preparada com os parâmetros fornecidos, sem aguardar pelos resultados.

```

int PQsendQueryPrepared(PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);

```

Semelhante à função `PQsendQueryParams`, mas o comando a ser executado é especificado pelo nome da declaração preparada anteriormente, em vez de fornecer a cadeia de caracteres do comando. Os parâmetros desta função são tratados de maneira idêntica aos da função `PQexecPrepared`. Da mesma forma que a função `PQexecPrepared`, não funciona em conexões com protocolo 2.0.

#### `PQgetResult`

Aguarda o próximo resultado de uma chamada anterior a `PQsendQuery`, `PQsendQueryParams`, `PQsendPrepare` ou `PQsendQueryPrepared`, e retorna o resultado. Retorna um ponteiro nulo quando o comando está completo e não haverá mais resultados.

```
PGresult *PQgetResult(PGconn *conn);
```

A função `PQgetResult` deve ser chamada repetidas vezes até que retorne um ponteiro nulo, indicando que o comando completou (Se for chamada quando nenhum comando estiver ativo, a função `PQgetResult` apenas retorna o ponteiro nulo de uma vez). Cada resultado não-nulo da função `PQgetResult` deve ser processado usando as mesmas funções de acesso a `PGresult` descritas anteriormente. Não se deve esquecer de liberar cada objeto de resultado através de `PQclear` após terminar de usá-lo. Deve ser observado que a função `PQgetResult` somente bloqueia se um comando estiver ativo, e os dados de resposta necessários ainda não foram lidos por `PQconsumeInput`.

A utilização da função `PQsendQuery` juntamente com a função `PQgetResult` resolve um dos problemas da função `PQexec`: Se uma cadeia de caracteres de comando incluir vários comandos SQL, os resultados destes comandos podem ser obtidos individualmente (Isto permite uma forma simples de processamento sobreposto, da seguinte maneira: o cliente pode estar tratando os resultados de um comando, enquanto o servidor ainda está trabalhando em comandos posteriores na mesma cadeia de caracteres de comando). Entretanto, chamar `PQgetResult` ainda faz com que o cliente fique bloqueado até que o servidor complete o próximo comando SQL. Isto pode ser evitado pelo uso apropriado de mais duas funções:

**PQconsumeInput**

Se estiver disponível uma entrada vinda do servidor, receber esta entrada.

```
int PQconsumeInput(PGconn *conn);
```

A função `PQconsumeInput` normalmente retorna 1, indicando “nenhum erro”, mas retorna 0 caso tenha acontecido algum problema (neste caso a função `PQerrorMessage` pode ser consultada). Deve ser observado que o resultado não informa se algum dado de entrada foi realmente coletado. Após chamar a função `PQconsumeInput`, o aplicativo pode verificar `PQisBusy` e/ou `PQnotifies` para ver se seu estado mudou.

A função `PQconsumeInput` pode ser chamada mesmo que o aplicativo ainda não esteja preparado para tratar o resultado ou a notificação. A função lê os dados disponíveis e guarda em um `buffer`, fazendo, portanto, com que uma indicação de pronto-para-ler da função `select()` desapareça. O aplicativo pode, portanto, utilizar a função `PQconsumeInput` para limpar a condição da função `select()` imediatamente, e depois examinar os resultados quando achar melhor.

**PQisBusy**

Retorna 1 se o comando estiver ocupado, ou seja, a função `PQgetResult` fica bloqueada aguardando pela entrada. O retorno do valor 0 indica que a função `PQgetResult` pode ser chamada com garantia que não vai bloquear.

```
int PQisBusy(PGconn *conn);
```

A função `PQisBusy` não tenta por si mesma ler os dados do servidor; portanto, a função `PQconsumeInput` deve ser chamada antes, ou o estado de ocupado nunca vai terminar.

Um aplicativo utilizando estas funções tipicamente tem um laço principal que usa a função `select()` ou `poll()` para aguardar por todas as condições que deve responder. Uma das condições é entrada disponível vinda do servidor, que em termos da função `select()` significa dados legíveis no descritor de arquivo identificado pela função `PQsocket`. Quando o laço principal detecta uma entrada pronta, deve chamar a função `PQconsumeInput` para ler a entrada. Depois pode chamar `PQisBusy`, seguida por `PQgetResult`, se `PQisBusy` retornar falso (0). Também pode chamar `PQnotifies` para detectar mensagens de NOTIFY (consulte a Seção 27.7).

Um cliente que utiliza as funções `PQsendQuery/PQgetResult` também pode tentar cancelar um comando que ainda está sendo processado pelo servidor; consulte a Seção 27.5. Mas a despeito do valor retornado por `PQcancel`, o aplicativo deve continuar com a seqüência normal de leitura de resultado utilizando `PQgetResult`. Um cancelamento bem-sucedido simplesmente faz com que o comando termine mais cedo do que faria de outra maneira.

Utilizando as funções descritas acima é possível evitar o bloqueio ao aguardar pela entrada vindo do servidor. Entretanto, ainda é possível que o aplicativo fique bloqueado aguardando para enviar a saída para o servidor. Isto é relativamente incomum, mas pode acontecer se forem enviados comandos SQL ou valores de dados muito longos (Entretanto, é muito mais provável quando o aplicativo envia dados através do comando `COPY IN`). Para evitar esta possibilidade, e obter uma operação com o banco de dados totalmente sem bloqueio, podem ser utilizadas as seguintes funções adicionais.

**PQsetnonblocking**

Define o status da conexão como bloqueante ou não.

```
int PQsetnonblocking(PGconn *conn, int arg);
```

Define o status da conexão como não bloqueante se `arg` for igual a 1, ou bloqueante se `arg` for igual a 0. Retorna 0 se for bem-sucedida, ou -1 se houver um erro.

No estado não bloqueante, a chamada a `PQsendQuery`, `PQputline`, `PQputnbytes` e `PQendcopy` não bloqueia, mas em vez disso retorna um erro se precisarem ser chamadas novamente.

Deve ser observado que a função `PQexec` não respeita o modo não bloqueante; se for chamada, age do modo bloqueante de qualquer maneira.

**PQisnonblocking**

Retorna o status da conexão com o banco de dados como bloqueante ou não.

```
int PQisnonblocking(const PGconn *conn);
```

Retorna 1 se a conexão estiver definida no modo não bloqueante, e 0 se bloqueante.

**PQflush**

Tenta descarregar qualquer dado de saída enfileirado para o servidor. Retorna 0 se for bem sucedido (ou se a fila de envio estiver vazia), -1 se falhar por alguma razão, ou 1 se não for capaz de enviar todos os dados ainda na fila de envio (este caso somente pode ocorrer se a conexão não for bloqueante).

```
int PQflush(PGconn *conn);
```

Após enviar qualquer comando ou dado por uma conexão não bloqueante, deve ser chamada a função `PQflush`. Se retornar 1, deve ser aguardado até que o soquete esteja pronto-para-escrita, e chamá-la novamente; repetir até que retorne 0. Uma vez que a função `PQflush` retorne 0, deve-se aguardar para que o soquete esteja pronto-para-leitura, e depois ler a resposta conforme descrito acima.

## 27.5. Cancelamento de comandos em andamento

Utilizando as funções descritas nesta seção, um aplicativo cliente pode solicitar o cancelamento de um comando que ainda está sendo processado pelo servidor.

**PQgetCancel**

Cria uma estrutura de dados contendo as informações necessárias para cancelar um comando enviado através de uma determinada conexão com o banco de dados.

```
PGcancel *PQgetCancel(PGconn *conn);
```

A função `PQgetCancel` cria o objeto `PGcancel` a partir do objeto de conexão `PGconn`. Retorna nulo se o parâmetro `conn` fornecido for nulo, ou se for uma conexão inválida. O objeto `PGcancel` é uma estrutura opaca que não foi feita para ser acessada diretamente pelo aplicativo; somente pode ser passado para a função `PQcancel` ou `PQfreeCancel`.

**PQfreeCancel**

Libera a estrutura de dados criada pela função `PQgetCancel`.

```
void PQfreeCancel(PGcancel *cancel);
```

A função `PQfreeCancel` libera o objeto de dados previamente criado pela função `PQgetCancel`.

**PQcancel**

Solicita ao servidor que abandone o processamento do comando corrente.

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

O valor retornado será igual a 1 quando o envio da solicitação de cancelamento for bem-sucedido, e 0 caso contrário. Se não for bem-sucedido, o parâmetro `errbuf` será preenchido com uma mensagem de erro explicando o motivo. O parâmetro `errbuf` deve ser uma matriz de caracteres do tamanho `errbufsize` (o tamanho recomendado é de 256 bytes).

Entretanto, o envio bem-sucedido não garante que a solicitação tenha algum efeito. Se o cancelamento for efetivo, o comando corrente termina mais cedo e retorna um resultado de erro. Se o cancelamento falhar (digamos, porque o servidor já tenha terminado de processar o comando), então não haverá nenhum resultado visível.

A função `PQcancel` pode ser chamada com segurança a partir de um tratador de sinais, se o parâmetro `errbuf` for uma variável local do tratador de sinais. No que diz respeito à função `PQcancel`, o objeto `PGcancel` é apenas para leitura, portanto esta função também pode ser chamada a partir de um fluxo de execução (`thread`) separado do fluxo de execução que manipula o objeto `PGconn`.

**PQrequestCancel**

Solicita ao servidor que abandone o processamento do comando corrente.

```
int PQrequestCancel(PGconn *conn);
```

A função `PQrequestCancel` é uma variante em obsolescência da função `PQcancel`. Esta função opera diretamente no objeto `PGconn`, e no caso de falha armazena a mensagem de erro no objeto `PGconn` (de onde a mensagem pode ser extraída pela função `PQerrorMessage`). Embora a funcionalidade seja a mesma, esta abordagem cria perigos para programas com vários fluxos de execução e para tratadores de sinais, por ser possível que a sobrescrita da mensagem de erro de `PGconn` atrapalhe a operação atualmente em andamento na conexão.

## 27.6. A interface de caminho-rápido

O PostgreSQL disponibiliza uma interface de caminho rápido (*fast-path*) para enviar chamadas simples de função para o servidor.

**Dica:** Esta interface está um tanto obsoleta, porque pode ser obtido um desempenho semelhante e uma funcionalidade melhor definindo uma declaração preparada para a chamada da função. Então, a execução da declaração com transmissão binária dos parâmetros e resultados substitui a chamada de função de caminho rápido.

A função `PQfn` solicita a execução da função do servidor através da interface de caminho rápido:

```
PGresult *PQfn(PGconn *conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);

typedef struct {
    int len;
    int isint;
    union {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;
```

O argumento `fnid` é o OID da função a ser executada. Os parâmetros `args` e `nargs` definem os parâmetros a serem passados para a função; estes parâmetros devem corresponder a lista declarada de argumentos da função. Quando o campo `isint` da estrutura do parâmetro tem o valor verdade, o valor de `u.integer` é enviado para o servidor como um inteiro do comprimento indicado (devendo ser 1, 2 ou 4 bytes); ocorre a troca apropriada de bytes. Quando `isint` tem o valor falso, o número de bytes indicados por `*u.ptr` é enviado sem processamento; os dados devem estar no formato esperado pelo servidor para transmissão binária do tipo de dado do argumento da função. O parâmetro `result_buf` é o buffer no qual é colocado o valor do retornado. Quem chama deve alocar espaço suficiente para armazenar o valor retornado (Não há verificação!) O real comprimento do resultado é retornado no inteiro apontado por `result_len`. Se for esperado um resultado inteiro de 1, 2 ou 4 bytes, `result_is_int` deve ser definido com o valor 1, senão com o valor 0. Definir `result_is_int` como 1 faz com que a libpq faça a troca de bytes do valor, caso seja necessário, para que este seja entregue como um valor do tipo `int` apropriado para a máquina cliente. Quando o valor de `result_is_int` é igual a 0, a cadeia de bytes no formato binário enviada pelo servidor é retornada sem modificações.

A função `PQfn` sempre retorna um ponteiro válido para `PGresult`. O status do resultado deve ser verificado antes do resultado ser utilizado. Quem chama é responsável por liberar `PGresult` através da função `PQclear`, quando esta não for mais necessária.

Deve ser observado que não é possível tratar argumentos nulos, resultados nulos, nem resultados de conjunto de valores nulos quando se utiliza esta interface.

## 27.7. Notificação assíncrona

O PostgreSQL disponibiliza notificação assíncrona através dos comandos `LISTEN` e `NOTIFY`. A sessão cliente registra seu interesse por uma determinada notificação através do comando `LISTEN` (e pára de ouvir através do comando `UNLISTEN`). Todas as sessões ouvindo uma determinada condição são notificadas assincronamente quando o comando `NOTIFY`, com este nome de condição, é executado por qualquer sessão. Não é passada nenhuma informação adicional para quem ouve. Portanto, qualquer dado que precise ser comunicado é transferido, usualmente, através de uma tabela do banco de dados. Normalmente, o nome da condição é o mesmo da tabela associada, mas não é necessário que haja nenhuma tabela associada.

Os aplicativos da libpq submetem os comandos `LISTEN` e `UNLISTEN` como comandos SQL comuns. A chegada das mensagens `NOTIFY` podem ser detectadas em seguida chamando a função `PQnotifies`.

A função `PQnotifies` retorna a próxima notificação de uma lista de mensagens de notificação não tratadas recebidas do servidor. Retorna um ponteiro nulo caso não haja mais notificações pendentes. Uma vez que a notificação seja retornada pela função `PQnotifies`, esta é considerada tratada e removida da lista de notificações.

```
PGnotify *PQnotifies(PGconn *conn);

typedef struct pgNotify {
    char *relname;           /* nome da condição de notificação */
    int be_pid;              /* ID de processo do processo servidor */
    char *extra;             /* parâmetro de notificação */
} PGnotify;
```

Após processar um objeto `PGnotify` retornado por `PQnotifies`, deve-se ter certeza que este é liberado através de `PQfreemem`. É suficiente liberar o ponteiro para `PGnotify`; os campos `relname` e `extra` não representam alocações separadas (No momento, o campo `extra` não é utilizado e sempre aponta para uma cadeia de caracteres vazia).

**Nota:** No PostgreSQL 6.4 e posteriores, o campo `be_pid` é relativo ao processo servidor fazendo a notificação, enquanto nas versões anteriores era sempre o PID do próprio processo servidor.

O Exemplo 27-2 mostra um programa exemplo que demonstra a utilização da notificação assíncrona.

Na verdade, a função `PQnotifies` não lê os dados do servidor; apenas retorna as mensagens previamente absorvidas por outra função da libpq. Nas versões anteriores da libpq, a única maneira de garantir a recepção a tempo das mensagens de NOTIFY era submetendo comandos constantemente, mesmo vazios, e depois verificando `PQnotifies` após cada `PQexec`. Embora isto ainda funcione, está em obsolescência e é um desperdício de poder de processamento.

Uma maneira melhor de verificar as mensagens de NOTIFY, quando não há comando útil a ser executado, é chamar `PQconsumeInput` e depois verificar `PQnotifies`. Pode ser utilizada a função `select()` para aguardar os dados chegarem do servidor, portanto usando ciclos da CPU a menos que haja algo a ser feito (Deve ser vista a função `PQsocket` para obter o número do descritor do arquivo a ser utilizado com a função `select()`). Deve ser observado que funciona bem se forem submetidos comandos através de `PQsendQuery/PQgetResult`, ou simplesmente utilizada a função `PQexec`. Entretanto, não se deve esquecer de verificar `PQnotifies` após cada `PQgetResult` ou `PQexec`, para ver se chegou alguma notificação durante o processamento do comando.

## 27.8. Funções associadas ao comando COPY

No PostgreSQL o comando `COPY` possui opções para ler ou escrever na conexão de rede utilizada pela libpq. As funções descritas nesta seção permitem que os aplicativos aproveitem as vantagens desta capacidade, fornecendo ou recebendo dados copiados.

No processamento global, primeiro o aplicativo envia um comando `COPY` do SQL através da função `PQexec`, ou através de uma função equivalente a esta. A resposta a este comando, se não houver erro, é um objeto `PGresult` contendo o código de status `PGRES_COPY_OUT` ou `PGRES_COPY_IN` (dependendo da direção especificada para a cópia). Depois, o aplicativo deve utilizar as funções descritas nesta seção para receber ou transmitir as linhas de dados. Quando a transferência de dados estiver completa, será retornado outro objeto `PGresult` indicando se a transferência foi bem ou mal-sucedida. O status será `PGRES_COMMAND_OK` para transferência bem-sucedida, ou `PGRES_FATAL_ERROR` caso ocorra algum problema. Neste ponto, podem ser enviados outros comandos SQL através da função `PQexec` (Não é possível executar outros comandos SQL utilizando a mesma conexão enquanto a operação de cópia estiver em andamento).

Se o comando `COPY` for executado através da função `PQexec` em uma cadeia de caracteres podendo conter outros comandos, o aplicativo deve continuar recebendo os resultados através da função `PQgetResult` após terminar a sequência do `COPY`. Somente quando a função `PQgetResult` retorna NULL tem-se certeza que a cadeia de caracteres de comando da função `PQexec` está concluída, e é seguro executar mais comandos.

As funções desta seção somente devem ser executadas após receber o status `PGRES_COPY_OUT` ou `PGRES_COPY_IN` no resultado de `PQexec` ou `PQgetResult`.

O objeto `PGresult` contendo um destes valores no status possui dados adicionais sobre a operação `COPY` iniciando. Este dados adicionais estão disponíveis através de funções também utilizadas com os resultados dos comandos na conexão:

**PQnfields**

Retorna o número de colunas (campos) a serem copiados.

**PQbinaryTuples**

0 indica que o formato global da cópia é textual (linhas separadas pelo caractere de nova-linha, colunas separadas pelo separador de caracteres, etc). 1 indica que o formato global da cópia é binário. Para obter informações adicionais deve ser consultado o comando *COPY*.

**PQfformat**

Retorna o código do formato (0 para texto, 1 para binário) associado com cada coluna da operação de cópia. Os códigos de formato por-coluna são sempre iguais a zero quando o formato global da cópia é textual, mas o formato binário permite tanto colunas binárias quanto texto (Entretanto, na implementação corrente do comando *COPY*, somente estão presentes colunas binárias na cópia binária; portanto, no momento os formatos por-coluna sempre correspondem ao formato global).

**Nota:** Estes valores de dados adicionais somente estão disponíveis quando se utiliza o protocolo 3.0. Quando se utiliza o protocolo 2.0, todas estas funções retornam 0.

### 27.8.1. Funções para enviar os dados do COPY

Estas funções são utilizadas para enviar os dados durante *COPY FROM STDIN*, e falham quando são chamadas em conexões que não se encontram no estado *COPY\_IN*.

**PQputCopyData**

Envia dados para o servidor durante o estado *COPY\_IN*.

```
int PQputCopyData(PGconn *conn,
                  const char *buffer,
                  int nbytes);
```

Transmite para o servidor os dados do *COPY* no *buffer* especificado, com comprimento *nbytes*. O resultado será igual a 1 se o dado for enviado, zero se não for enviado porque a tentativa bloquearia (este caso somente é possível se a conexão for no modo não bloqueante), ou -1 caso ocorra um erro (Quando o valor retornado for igual a -1 deverá ser utilizada a função *PQerrorMessage* para obter detalhes. Se o valor retornado for igual a zero, deve-se aguardar por pronto-para-escrever e tentar novamente).

O aplicativo pode dividir o fluxo de dados do *COPY* em cargas de *buffer* de qualquer tamanho conveniente. As fronteiras do *buffer* de carga não possuem significado semântico ao enviar. O conteúdo do fluxo de dados deve corresponder ao formato esperado pelo comando *COPY*; para obter detalhes deve ser visto o comando *COPY*.

**PQputCopyEnd**

Envia a indicação de fim de dados para o servidor durante o estado *COPY\_IN*.

```
int PQputCopyEnd(PGconn *conn,
                 const char *errmsg);
```

Termina a operação *COPY\_IN* com sucesso se *errmsg* for *NULL*. Se *errmsg* não for *NULL*, então o *COPY* é obrigado a falhar, com a cadeia de caracteres apontada por *errmsg* usada como mensagem de erro (Entretanto, não se deve assumir que virá do servidor uma determinada mensagem, porque o servidor pode falhar na execução do comando *COPY* por seus próprios motivos. Também deve ser observado que forçar a falha não funciona quando se utiliza conexões com protocolo pré-3.0).

O resultado será igual a 1 se o dado de terminação for enviado, zero se não for enviado porque a tentativa bloquearia (este caso somente é possível se a conexão for no modo não bloqueante), ou -1 caso ocorra um erro (Quando o valor retornado for igual a -1 deverá ser utilizada a função *PQerrorMessage* para obter detalhes. Se o valor retornado for igual a zero, deve-se aguardar por pronto-para-escrever e tentar novamente).

Após uma chamada bem-sucedida à função *PQputCopyEnd*, deve ser chamada a função *PQgetResult* para obter o status final do resultado do comando *COPY*. Deve-se aguardar por este resultado da maneira usual. Em seguida voltar à operação normal.

### 27.8.2. Funções para receber os dados do COPY

Estas funções são utilizadas para receber os dados durante `COPY TO STDOUT`, e falham quando são chamadas em conexões que não se encontram no estado `COPY_OUT`.

#### PQgetCopyData

Recebe dados do servidor durante o estado `COPY_OUT`.

```
int PQgetCopyData(PGconn *conn,
                  char **buffer,
                  int async);
```

Tenta obter outra linha de dados do servidor durante o `COPY`. Os dados são sempre retornados uma linha de cada vez; se estiver disponível apenas uma linha parcial, esta não é retornada. O retorno bem-sucedido da linha de dados envolve a alocação de um bloco de memória para conter os dados. O parâmetro `buffer` deve ser diferente de `NULL`. O parâmetro `*buffer` é definido para apontar para a memória alocada, ou para `NULL` nos casos em que nenhum `buffer` é retornado. Um `buffer` de resultado diferente de `NULL` deve ser liberado através da função `PQfreemem` quando não for mais necessário.

Quando uma linha é retornada com sucesso, o valor retornado é o número de bytes de dados na linha (sempre será maior que zero). A cadeia de caracteres retornada é sempre terminada por nulo, embora provavelmente somente será útil para o comando `COPY` no formato texto. Um resultado igual a zero indica que o comando `COPY` ainda se encontra em andamento, mas que nenhuma linha está disponível no momento (isto somente é possível quando o parâmetro `async` é igual a verdade). O resultado com valor -1 indica que o comando `COPY` chegou ao fim. O resultado -2 indica que ocorreu um erro (deve ser consultada a função `PQerrorMessage` para saber o motivo).

Quando o valor do parâmetro `async` é igual a verdade (diferente de zero), a função `PQgetCopyData` não bloqueia aguardando pela entrada, e retorna zero se o comando `COPY` ainda estiver em andamento mas não existir nenhuma linha completa disponível (Neste caso deve-se aguardar por pronto-para-ler antes de tentar novamente; não importa se foi chamada a função `PQconsumeInput`). Quando `async` é falso (zero), a função `PQgetCopyData` bloqueia até que haja dados disponíveis, ou que a operação complete.

Após a função `PQgetCopyData` retornar o valor -1, deve-se chamar a função `PQgetResult` para obter o status do resultado final do comando `COPY`. Deve-se aguardar por este resultado da maneira usual. Em seguida voltar à operação normal.

### 27.8.3. Funções obsoletas para o COPY

Estas funções representam métodos antigos para tratar o comando `COPY`. Embora ainda funcionem, estão em obsolescência devido a um fraco tratamento de erros, métodos inconvenientes para detectar o fim dos dados, e falta de suporte para transferências binárias e não bloqueantes.

#### PQgetline

Lê uma linha terminada pelo caractere nova-linha (transmitida pelo servidor) em um `buffer` cadeia de caracteres com comprimento `length`.

```
int PQgetline(PGconn *conn,
              char *buffer,
              int length);
```

Esta função copia até `length-1` caracteres para o `buffer` e converte o caractere nova-linha terminador em um byte zero. A função `PQgetline` retorna EOF no final da entrada, 0 se toda a linha já foi lida, e 1 se o `buffer` estiver cheio mas o caractere nova-linha terminador ainda não foi lido.

Deve ser observado que o aplicativo precisa verificar se a nova linha consiste nos dois caracteres `\.`, indicando que o servidor terminou de enviar os resultados do comando `COPY`. Se o aplicativo puder receber linhas com comprimento maior que `length-1` caracteres, deve-se tomar o cuidado que este reconheça `\.` corretamente (e, por exemplo, não confunda o final de uma linha de dados longa com a linha terminadora).

#### PQgetlineAsync

Lê uma linha de dados do `COPY` (transmitida pelo servidor) em um `buffer` sem bloquear.

```
int PQgetlineAsync(PGconn *conn,
```

```
char *buffer,
int bufsize);
```

Esta função é semelhante à função `PQgetline`, mas pode ser utilizada por aplicativos que devem ler os dados do `COPY` assincronamente, ou seja, sem bloquear. Uma vez emitido o comando `COPY` e obtida uma resposta `PGRES_COPY_OUT`, o aplicativo deve chamar `PQconsumeInput` e `PQgetlineAsync` até que o sinal de fim dos dados seja detectado.

Ao contrário de `PQgetline`, esta função recebe a responsabilidade de detectar o fim dos dados.

A cada chamada, `PQgetlineAsync` retorna dados se estiver disponível no `buffer` de entrada da `libpq` uma linha de dados completa. Senão, nenhum dado é retornado até que o restante da linha chegue. A função retorna -1 quando a marca de fim-de-dados-de-cópia (`end-of-copy-data`) é reconhecida, ou 0 quando não há dados disponíveis, ou um número positivo indicando o número de bytes de dados retornados. Se for retornado -1, quem chamou deve chamar `PQendcopy` em seguida, e depois retornar ao processamento normal.

Os dados retornados não vão além da fronteira dos dados da linha. Se for possível, é retornada uma linha inteira de cada vez. Mas se o `buffer` oferecido por quem chama for muito pequeno para guardar uma linha enviada pelo servidor, então são retornados dados parciais da linha. Com dados no formato texto isto pode ser detectado testando se o último byte retornado é `\n`, ou não (Em um comando `COPY` binário, será necessária a análise do formato real dos dados do `COPY` para fazer uma determinação equivalente). A cadeia de caracteres retornada não é terminada por nulo (Se for desejado adicionar um nulo terminador, deve-se ter certeza de passar `bufsize` com tamanho de um caractere a menos que espaço realmente disponível).

#### `PQputline`

Envia uma cadeia de caracteres terminada por nulo para o servidor. Retorna 0 quando é bem-sucedida, e EOF quando não consegue enviar a cadeia de caracteres.

```
int PQputline(PGconn *conn,
              const char *string);
```

O fluxo de dados do `COPY` enviado por uma série de chamadas à função `PQputline` possui o mesmo formato que o retornado pela função `PQgetlineAsync`, exceto que os aplicativos não são obrigados a enviar uma linha de dados por chamada à função `PQputline`; não há problema em se enviar uma linha parcial, ou várias linhas por chamada.

**Nota:** Antes do protocolo 3.0 do PostgreSQL, era necessário que o aplicativo enviasse explicitamente os dois caracteres `\.` como linha final para indicar ao servidor que tinha terminado de enviar os dados do comando `COPY`. Embora isto ainda funcione, está em obsolescência e pode ser esperado que seja removido o significado especial de `\.` em uma versão futura. Basta chamar a função `PQendcopy` após ter enviado os dados.

#### `PQputnbytes`

Envia uma cadeia de caracteres não terminada por nulo para o servidor. Retorna 0 quando é bem-sucedida, e EOF quando não consegue enviar a cadeia de caracteres.

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

É exatamente igual a `PQputline`, exceto que o `buffer` dos dados não precisa ser terminado por nulo, uma vez que o número de bytes a serem enviados é especificado diretamente. Deve ser utilizada esta função quando se envia dados binários.

#### `PQendcopy`

Sincroniza com o servidor.

```
int PQendcopy(PGconn *conn);
```

Esta função aguarda até que o servidor tenha terminado de copiar. Deve ser chamada quando a última cadeia de caracteres já tiver sido enviada para o servidor utilizando `PQputline`, ou quando a última cadeia de caracteres tiver sido recebida do servidor utilizando `PQgetline`. Deve ser chamada ou o servidor ficará “fora de sincronia” com o cliente. Após o retorno desta função, o servidor está pronto para receber o próximo comando SQL. Retorna o valor 0 ao término bem-sucedido, ou um valor diferente de zero caso contrário (Quando o valor retornado for diferente de zero, deve ser utilizada a função `PQerrorMessage` para obter detalhes).



Quando se usa `PQgetResult`, o aplicativo deve responder ao resultado `PGRES_COPY_OUT` executando `PQgetline` repetidamente, seguida por `PQendcopy` após ter encontrado a linha terminadora. Depois deve voltar ao laço `PQgetResult` até que `PQgetResult` retorne um ponteiro nulo. De forma semelhante, um resultado `PGRES_COPY_IN` é processado por uma série de chamadas a `PQputline` seguida por `PQendcopy`, e depois retornar ao laço `PQgetResult`. Este arranjo garante que o comando `COPY` incorporado a uma série de comandos SQL será executado de maneira correta.

Os aplicativos antigos provavelmente enviam o comando `COPY` através da função `PQexec` e assumem que a transação está terminada após `PQendcopy`. Isto funciona de maneira correta somente se o comando `COPY` for o único comando SQL na cadeia de caracteres do comando.

## 27.9. Funções de controle

Estas funções controlam diversos detalhes de comportamento da libpq.

`PQsetErrorVerbosity`

Determina a verbosidade das mensagens retornadas pelas funções `PQerrorMessage` e `PQresultErrorMessage`.

```
typedef enum {
    PQERRORS_TERSE,
    PQERRORS_DEFAULT,
    PQERRORS_VERBOSE
} PGVerbosity;
```

```
PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity verbosity);
```

A função `PQsetErrorVerbosity` define o modo de verbosidade, e retorna a definição anterior da conexão. No modo *TERSE* (sucinto), as mensagens retornadas incluem apenas a severidade, o texto primário e a posição; normalmente cabe em apenas uma linha. O modo padrão produz mensagens que incluem os itens acima mais os campos detalhe, dica e contexto (pode abranger várias linhas). O modo *VERBOSE* inclui todos os campos disponíveis. Mudar a verbosidade não afeta as mensagens disponíveis nos objetos `PGresult` já existentes, somente afeta os objetos criados após a mudança.

`PQtrace`

Habilita o envio do rastreamento da comunicação cliente/servidor para um arquivo de depuração.

```
void PQtrace(PGconn *conn, FILE *stream);
```

`PQuntrace`

Desabilita o rastreamento iniciado por `PQtrace`.

```
void PQuntrace(PGconn *conn);
```

## 27.10. Processamento de notas

As mensagens de nota e advertência geradas pelo servidor não são retornadas pelas funções que executam os comandos, uma vez que não implicam em falha do comando. Em vez disso, são passadas para uma função tratadora, e a execução prossegue normalmente após o retorno do tratador. A função padrão para tratar notas envia a mensagem para `stderr`, mas o aplicativo pode mudar este comportamento fornecendo sua própria função tratadora.

Por motivos históricos existem dois níveis de tratamento de notas, chamados receptor de notas e processador de notas. O comportamento padrão é o receptor formatar a nota e passar a cadeia de caracteres para o processador de notas, para que este faça a exibição. Entretanto, um aplicativo que decida fornecer seu próprio receptor de notas, tipicamente ignora a camada do processador de notas e apenas realiza todo o trabalho no receptor de notas.

A função `PQsetNoticeReceiver` define ou consulta o receptor de notas corrente para o objeto de conexão. De maneira semelhante, a função `PQsetNoticeProcessor` define ou consulta o processador de notas corrente.

```
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);
```

`PQnoticeReceiver`

```
PQsetNoticeReceiver(PGconn *conn,
                   PQnoticeReceiver proc,
```

```

        void *arg);

typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                    PQnoticeProcessor proc,
                    void *arg);

```

Estas funções retornam o ponteiro para a função receptora de notas ou processadora de notas anterior, e definem o novo valor. Se for fornecido um ponteiro de função nulo nenhuma ação é realizada, mas é retornado o ponteiro corrente.

Quando é recebida uma mensagem de nota ou advertência vinda do servidor, ou gerada internamente pela libpq, a função receptora de notas é chamada. A mensagem é passada na forma de um `PGresult PGRES_NONFATAL_ERROR` (Permite ao receptor extrair os campos individualmente utilizando a função `PQresultErrorField`, ou a mensagem completa pré-formatada utilizando a função `PQresultErrorMessage`). O mesmo ponteiro vazio passado para a função `PQsetNoticeReceiver` também é passado (Este ponteiro pode ser utilizado para acessar estados específicos do aplicativo, caso haja necessidade).

O receptor padrão de notas simplesmente extrai a mensagem (utilizando a função `PQresultErrorMessage`), e passa para o processador de notas.

O processador de notas é responsável por tratar as mensagens de nota e advertência fornecidas na forma de texto. É passada a cadeia de caracteres do texto da mensagem (incluindo um caractere de nova-linha no final), mais um ponteiro vazio que é o mesmo passado para a função `PQsetNoticeProcessor` (Este ponteiro pode ser utilizado para acessar estados específicos do aplicativo, caso haja necessidade).

O processador de notas padrão é simplesmente

```

static void
defaultNoticeProcessor(void *arg, const char *message)
{
    fprintf(stderr, "%s", message);
}

```

Uma vez definido um receptor ou processador de notas, deve-se esperar que sejam chamados enquanto os objetos `PGconn` ou `PGresult` que acessam os mesmos existam. Na criação de um objeto `PGresult`, os ponteiros para o tratador de notas corrente de `PGconn` são copiados para `PGresult`, para um possível uso por parte de funções como `PQgetvalue`.

## 27.11. Variáveis de Ambiente

As seguintes variáveis de ambiente podem ser utilizadas para selecionar o valor padrão dos parâmetros de conexão a serem utilizados pelas funções `PQconnectdb`, `PQsetdbLogin` e `PQsetdb`, se não for especificado nenhum valor no código que faz a chamada. É útil para evitar prender as informações de conexão com o banco de dados ao código dos aplicativos cliente, por exemplo.

- `PGHOST` define o nome do servidor de banco de dados. Se começar por uma barra, especifica uma comunicação no domínio Unix em vez de uma comunicação TCP/IP; o valor é então o nome do diretório no qual o arquivo de soquete é armazenado (na configuração padrão de instalação seria `/tmp`).
- `PGHOSTADDR` especifica o endereço numérico de IP do servidor de banco de dados. Pode ser definido adicionalmente a `PGHOST`, para evitar o trabalho extra de procura no DNS. Para obter detalhes sobre como interagem, deve ser consultada a documentação relativa aos parâmetros da função `PQconnectdb` acima.

Quando não é definido `PGHOST` nem `PGHOSTADDR`, o comportamento padrão é conectar utilizando o soquete do domínio Unix local; em máquinas sem soquetes do domínio Unix, a libpq tenta a conexão com `localhost`.

- `PGPORT` define o número da porta TCP, ou a extensão do arquivo de soquete do domínio Unix para a comunicação com o servidor PostgreSQL.
- `PGDATABASE` define o nome do banco de dados do PostgreSQL.
- `PGUSER` define o nome de usuário utilizado para conectar ao banco de dados.

- `PGPASSWORD` define a senha utilizada se o servidor requerer autenticação por senha. Esta variável de ambiente está em obsolescência por motivos de segurança; em seu lugar deve ser considerado o uso do arquivo `~/.pgpass` (consulte a Seção 27.12).
- `PGSERVICE` define o nome do serviço a ser procurado em `pg_service.conf`. Oferece uma forma abreviada para definir todos os parâmetros.
- `PGREALM` define o `realm` do Kerberos a ser utilizado com o PostgreSQL, se for diferente do `realm` local. Se a variável de ambiente `PGREALM` for definida, os aplicativos da libpq tentam a autenticação com os servidores para este `realm`, e utilizam arquivos de tíquete separados para evitar conflito com arquivos de tíquete locais. Esta variável de ambiente somente é utilizada quando é escolhida pelo servidor a autenticação Kerberos.
- `PGOPTIONS` define opções adicionais em tempo de execução para o servidor PostgreSQL.
- `PGSSLMODE` determina se, e com que prioridade, será negociada uma conexão SSL com o servidor. Existem quatro modos: `disable` tenta apenas uma conexão SSL não criptografada; `allow` negocia, tentando primeiro uma conexão não-SSL e depois, se falhar, tenta uma conexão SSL; `prefer` (o padrão) negocia, tentando primeiro uma conexão SSL e depois, se falhar, uma conexão normal não-SSL; `require` somente tenta uma conexão SSL. Se o PostgreSQL for compilado sem suporte a SSL, `require` causa um erro, enquanto as opções `allow` e `prefer` são aceitas, mas na verdade a libpq não tenta uma conexão SSL.
- `PGREQUIRESSL` define se a conexão deve ser feita através da SSL, ou não. Se for definida como “1”, a libpq recusa a conexão se o servidor não aceitar uma conexão SSL (equivale a `sslmode prefer`). Esta opção está em obsolescência em favor da definição de `sslmode`, e somente está disponível quando o PostgreSQL é compilado com suporte a SSL.
- `PGCONNECT_TIMEOUT` define o número máximo de segundos que a libpq aguarda na tentativa de conectar ao servidor PostgreSQL. Se não estiver definida, ou se for definida como zero, a libpq aguarda indefinidamente. Não se recomenda colocar o tempo para ficar aguardando inferior a 2 segundos.

As seguintes variáveis de ambiente podem ser utilizadas para especificar o comportamento padrão para cada sessão do PostgreSQL (Também devem ser vistas as maneiras de se definir o comportamento padrão por usuário e por banco de dados nos comandos `ALTER USER` e `ALTER DATABASE`).

- `PGDATESTYLE` define o estilo padrão para a representação de data/hora (Equivale a `SET datestyle TO ...`).
- `PGTZ` define a zona horária padrão. (Equivale a `SET timezone TO ...`).
- `PGCLIENTENCODING` define a codificação padrão do conjunto de caracteres do cliente (Equivale a `SET client_encoding TO ...`).
- `PGGEQO` define o modo padrão para o otimizador de comandos genético (Equivale a `SET geqo TO ...`).

Para obter informações sobre os valores corretos destas variáveis de ambiente, deve ser visto o comando `SET` do SQL.

As seguintes variáveis de ambiente determinam o comportamento interno da libpq; elas substituem os padrões de compilação.

- `PGSYSCONFDIR` define o diretório que contém o arquivo `pg_service.conf`.
- `PGLOCALEDIR` define o diretório que contém os arquivos `locale` para as mensagens de internacionalização.

## 27.12. O arquivo de senhas

O arquivo `.pgpass`, armazenado na pasta base (`home`) do usuário, é um arquivo que contém senhas a serem utilizadas se a conexão requisitar uma senha (e a senha não tiver sido especificada de outra maneira). No Microsoft Windows o arquivo se chama `%APPDATA%\postgresql\pgpass.conf` (onde `%APPDATA%` se refere ao subdiretório de dados do aplicativo no perfil do usuário).

Este arquivo deve conter linhas com o seguinte formato:

```
nome_do_hospedeiro:porta:nome_do_banco_de_dados:nome_do_usuario:senha
```

Os quatro primeiros valores podem ser um literal, ou `*` para corresponder a qualquer coisa. É utilizada a senha da primeira linha que corresponder aos parâmetros da conexão corrente (portanto, as entradas mais específicas devem ser colocadas primeiro quando são utilizados curingas). Se a entrada precisar conter os caracteres `:` ou `\`, estes caracteres devem receber o escape de `\`.

As permissões de acesso ao arquivo `.pgpass` não devem permitir o acesso por todos os usuários ou para grupos; isto é conseguido pelo comando `chmod 0600 ~/.pgpass`. Se as permissões forem mais rígidas que esta, o arquivo será ignorado (Entretanto, atualmente as permissões não são verificadas no Microsoft Windows).

## 27.13. Suporte a SSL

Para aumentar a segurança, o PostgreSQL possui suporte nativo ao uso de conexões SSL para criptografar a comunicação cliente/servidor. Para obter detalhes sobre as funcionalidades do SSL do lado servidor, deve ser vista a Seção 16.7.

Se o servidor requisitar um certificado do cliente, a libpq envia o certificado presente no arquivo `~/.postgresql/postgresql.crt` armazenado na pasta base do usuário. O arquivo de chave privada correspondente `~/.postgresql/postgresql.key` também deve estar presente, e não pode ser legível por todos os usuários (No Microsoft Windows estes arquivos se chamam `%APPDATA%\postgresql\postgresql.crt` e `%APPDATA%\postgresql\postgresql.key`).

Se o arquivo `~/.postgresql/root.crt` estiver presente no diretório base do usuário, a libpq utiliza a lista de certificados armazenada neste arquivo para verificar o certificado do servidor (No Microsoft Windows o arquivo se chama `%APPDATA%\postgresql\root.crt`). A conexão SSL falha se o servidor não apresentar o certificado; portanto, para utilizar esta funcionalidade o servidor também deve possuir um arquivo `root.crt`.

## 27.14. Comportamento dos programas com fluxo de execução

A libpq é reentrante e segura quanto a fluxos de execução (`thread-safe`), quando é utilizada a opção de linha de comando `--enable-thread-safety` do `configure` na construção da distribuição do PostgreSQL. Além disso, ao se compilar o código do aplicativo, podem ser necessárias opções adicionais de linha de comando do compilador. Para obter informações sobre como construir aplicativos com fluxos de execução deve ser consultada a documentação do sistema, ou procurar no arquivo `src/Makefile.global` por `PTHREAD_CFLAGS` e `PTHREAD_LIBS`.

Uma restrição é que dois fluxos não devem tentar manipular o mesmo objeto `PGconn` ao mesmo tempo. Em particular, não é possível emitir comandos concorrentes, através do mesmo objeto de conexão, a partir de fluxos de execução diferentes (Se for necessário executar comandos simultâneos, devem ser utilizadas várias conexões).

Os objetos `PGresult`, após serem criados, são apenas para leitura e, portanto, podem ser passados livremente entre os fluxos.

As funções em obsolescência `PQrequestCancel`, `PQoidStatus` e `fe_setauthsvc` não são seguras quanto a fluxos de execução, não devendo ser utilizadas em programas com vários fluxos de execução. A função `PQrequestCancel` pode ser substituída pela função `PQcancel`. A função `PQoidStatus` pode ser substituída pela função `PQoidValue`. Na verdade, não há nenhuma boa razão para se chamar a função `fe_setauthsvc`.

Os aplicativos libpq que utilizam o método de autenticação `crypt` dependem da função `crypt()` do sistema operacional, que geralmente não é segura quanto a fluxos de execução. É melhor utilizar o método `md5`, que é seguro quanto a fluxos de execução em todas as plataformas.

Ocorrendo problemas em aplicativos com fluxos de execução, deve ser executado o programa `src/tools/thread` para verificar se a plataforma utilizada possui funções não seguras quanto a fluxos de execução. Este programa é executado pelo `configure`, mas para as distribuições binárias a biblioteca sendo utilizada pode não corresponder à biblioteca utilizada para construir os binários.

## 27.15. Construção de programas que utilizam a libpq

Para construir (ou seja, compilar e ligar) um programa que utiliza a libpq, é necessário realizar as seguintes atividades:

- Incluir o arquivo de cabeçalho `libpq-fe.h`:

```
#include <libpq-fe.h>
```

Quando isto não é feito, normalmente são recebidas mensagens de erro do compilador semelhantes a:

```
foo.c: In function `main':
foo.c:34: `PGconn' undeclared (first use in this function)
foo.c:35: `PGresult' undeclared (first use in this function)
foo.c:54: `CONNECTION_BAD' undeclared (first use in this function)
```

```
foo.c:68: `PGRES_COMMAND_OK' undeclared (first use in this function)
foo.c:95: `PGRES_TUPLES_OK' undeclared (first use in this function)
```

- Adicionar o diretório onde os arquivos de cabeçalho do PostgreSQL estão armazenados à lista de diretórios procurados, fornecendo a opção `-Idiretório` para o compilador (Em alguns casos o compilador procura o diretório em questão por padrão, podendo-se omitir esta opção). Por exemplo, a linha de comando do compilador pode se parecer com:

```
cc -c -I/usr/local/pgsql/include testprog.c
```

Se estiver sendo utilizado o arquivo Makefile, então a opção deve ser adicionada à variável CPPFLAGS:

```
CPPFLAGS += -I/usr/local/pgsql/include
```

Havendo possibilidade do programa ser compilado por outros usuários, então o local do diretório não deve ser fixado desta maneira. Em vez disso, pode ser executado o utilitário `pg_config` para descobrir onde estão os arquivos de cabeçalho no sistema local:

```
$ pg_config --includedir
/usr/local/include
```

A falta da especificação correta desta opção para o compilador resulta em uma mensagem de erro do tipo:

```
testlibpq.c:8:22: libpq-fe.h: Arquivo ou diretório não encontrado
```

- Ao se ligar o programa final, deve ser especificada a opção `-lpq` para que a biblioteca libpq seja procurada na ligação, assim como a opção `-Ldiretório` para adicionar o diretório onde a biblioteca libpq reside à lista de diretórios a serem procurados por `-l` (Novamente, o compilador procura em alguns diretórios por padrão). Para o máximo de portabilidade, a opção `-L` deve ser colocada antes da opção `-lpq`. Por exemplo:

```
cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq
```

O diretório da biblioteca pode ser descoberto utilizando `pg_config` também:

```
$ pg_config --libdir
/usr/local/pgsql/lib
```

As mensagens de erro apontando problemas nesta área se parecem com o seguinte.

```
testlibpq.o(.text+0xd): In function `exit_nicely':
: undefined reference to `PQfinish'
testlibpq.o(.text+0x5b): In function `main':
: undefined reference to `PQconnectdb'
testlibpq.o(.text+0x6c): In function `main':
: undefined reference to `PQstatus'
...
```

Isto significa que `-lpq` foi esquecido.

```
/usr/bin/ld: cannot find -lpq
```

Isto significa que a opção `-L` foi esquecida, ou que não foi especificado o diretório correto.

Se o código fizer referência ao arquivo de cabeçalho `libpq-int.h`, e você se recusa a corrigir o código para que não faça mais, do PostgreSQL 7.2 em diante este arquivo pode ser encontrado em `includedir/postgresql/internal/libpq-int.h`, portanto será necessário acrescentar a opção `-I` apropriada à linha de comando do compilador.

## 27.16. Programas exemplo

Estes exemplos, e outros, podem ser encontrados no diretório `src/test/examples` na distribuição do código fonte.

### Exemplo 27-1. Programa exemplo da libpq nº 1

```
/*
 * testlibpq.c
 *
 * Testa a versão C da LIBPQ, a biblioteca de interface com o POSTGRES
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"
```

```

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char    *conninfo;
    PGconn        *conn;
    PGresult      *res;
    int            nFields;
    int            i,
                  j;

    /*
     * Se o usuário fornecer o parâmetro na linha de comando, este é
     * utilizado na cadeia de caracteres conninfo; senão, o padrão é
     * definir dbname=templatel e utilizar as variáveis de ambiente,
     * ou o valor padrão, para todos os outros parâmetros de conexão.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = templatel";

    /* Realizar a conexão com o banco de dados */
    conn = PQconnectdb(conninfo);

    /* Verificar se a conexão com o servidor foi bem-sucedida */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "A conexão com o banco de dados falhou: %s",
            PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /*
     * O caso de teste envolve a utilização de um cursor, motivo pelo qual
     * é necessário estar dentro de um bloco de transação. Tudo poderia
     * ser feito com uma única execução de "select * from pg_database"
     * utilizando a função PQexec(), mas seria muito trivial para servir
     * como um bom exemplo.
     */

    /* Iniciar o bloco de transação */
    res = PQexec(conn, "BEGIN");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "O comando BEGIN falhou: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * Para evitar vazamento de memória é necessário chamar PQclear PGresult
     * sempre que este não é mais necessário.
     */
    PQclear(res);

```

```

/*
 * Trazer as linhas de pg_database, o catálogo de bancos de dados
 */
res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "DECLARE CURSOR falhou: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

res = PQexec(conn, "FETCH ALL IN myportal");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "FETCH ALL falhou: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/* primeiro, imprimir os nomes dos atributos */
nFields = PQnfields(res);
for (i = 0; i < nFields; i++)
    printf("%-15s", PQfname(res, i));
printf("\n");

/* em seguida, imprimir as linhas */
for (i = 0; i < PQntuples(res); i++)
{
    for (j = 0; j < nFields; j++)
        printf("%-15s", PQgetvalue(res, i, j));
    printf("\n");
}

PQclear(res);

/* fechar o portal ... sem se importar com a verificação de erros ... */
res = PQexec(conn, "CLOSE myportal");
PQclear(res);

/* encerrar a transação */
res = PQexec(conn, "END");
PQclear(res);

/* fechar a conexão com o banco de dados e limpar */
PQfinish(conn);

return 0;
}

```

Execução do programa com o comando SQL mudado para “DECLARE myportal CURSOR FOR select datname, datistemplate, datallowconn from pg\_database where datdba=1”.<sup>3</sup>

```

$ gcc testlibpq.c -o testlibpq \
> -I /usr/local/pgsql/include/ \
> -L /usr/local/pgsql/lib/ -lpq
$ ./testlibpq "host=localhost user=teste password=teste dbname=teste"

```

datname	datistemplate	datallowconn
templatel	t	t
template0	t	f

**Exemplo 27-2. Programa exemplo da libpq nº 2**

```

/*
 * testlibpq2.c
 *
 *      Teste da interface de notificação assíncrona
 *
 * Este programa deve ser iniciado e, depois, usando o psql em outra janela
 * deve ser executado:
 *   NOTIFY TBL2;
 * Deve ser repetido quatro vezes para que este programa termine.
 *
 * Ou, para aprimorar, deve-se tentar:
 * carregar o banco de dados com os seguintes comandos
 * (presentes no arquivo src/test/examples/testlibpq2.sql):
 *
 *   CREATE TABLE TBL1 (i int4);
 *
 *   CREATE TABLE TBL2 (i int4);
 *
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
 *
 * e executar este comando quatro vezes:
 *
 *   INSERT INTO TBL1 VALUES (10);
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult    *res;
    PGnotify    *notify;
    int         nnotifies;

    /*
     * Se o usuário fornecer o parâmetro na linha de comando, este é
     * utilizado na cadeia de caracteres conninfo; senão, o padrão é
     * definir dbname=templatel e utilizar as variáveis de ambiente,
     * ou o valor padrão, para todos os outros parâmetros de conexão.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = templatel";

    /* Realizar a conexão com o banco de dados */

```



```

conn = PQconnectdb(conninfo);

/* Verificar se a conexão com o servidor foi bem-sucedida */
if (PQstatus(conn) != CONNECTION_OK)
{
    fprintf(stderr, "A conexão com o banco de dados falhou: %s",
        PQerrorMessage(conn));
    exit_nicely(conn);
}

/*
 * Emitir o comando LISTEN para habilitar as notificações
 * das regras de NOTIFY.
 */
res = PQexec(conn, "LISTEN TBL2");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "O comando LISTEN falhou: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/*
 * Para evitar vazamento de memória é necessário chamar PQclear PGresult
 * sempre que este não é mais necessário.
 */
PQclear(res);

/* Sair após receber quatro notificações. */
nnotifies = 0;
while (nnotifies < 4)
{
    /*
     * Adormecer até que alguma coisa aconteça na conexão.
     * É utilizado select(2) para aguardar pela entrada, mas
     * também poderia ser utilizado poll() ou algo semelhante.
     */
    int sock;
    fd_set input_mask;

    sock = PQsocket(conn);

    if (sock < 0)
        break; /* não deve acontecer */

    FD_ZERO(&input_mask);
    FD_SET(sock, &input_mask);

    if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
    {
        fprintf(stderr, "select() falhou: %s\n", strerror(errno));
        exit_nicely(conn);
    }

    /* Verificar a entrada */
    PQconsumeInput(conn);
    while ((notify = PQnotifies(conn)) != NULL)
    {
        fprintf(stderr,
            "ASYNC NOTIFY '%s' recebida do servidor com pid %d\n",
            notify->relname, notify->be_pid);
        PQfreemem(notify);
    }
}

```

```

        nnotifies++;
    }
}

fprintf(stderr, "Terminado.\n");

/* fechar a conexão com o banco de dados e limpar */
PQfinish(conn);

return 0;
}

```

Execução do programa exemplo:<sup>4</sup>

```

$ gcc testlibpq2.c -o testlibpq2 \
> -I /usr/local/pgsql/include/ \
> -L /usr/local/pgsql/lib/ -lpq
$ ./testlibpq2

```

```

ASYNC NOTIFY 'tbl2' recebida do servidor com pid 5685
ASYNC NOTIFY 'tbl2' recebida do servidor com pid 5685
ASYNC NOTIFY 'tbl2' recebida do servidor com pid 5685
ASYNC NOTIFY 'tbl2' recebida do servidor com pid 5685
Terminado.

```

### Exemplo 27-3. Programa exemplo da libpq nº 3

```

/*
 * testlibpq3.c
 *
 *      Teste de parâmetros fora-de-linha e E/S binária
 *
 * Antes de executar este exemplo, o banco de dados deve ser carregado
 * com os seguintes comandos
 * (fornecidos no arquivo src/test/examples/testlibpq3.sql):
 *
 * CREATE TABLE test1 (i int4, t text, b bytea);
 *
 * INSERT INTO test1 values (1, 'joe's place', '\\000\\001\\002\\003\\004');
 * INSERT INTO test1 values (2, 'ho there', '\\004\\003\\002\\001\\000');
 *
 * A saída esperada é:
 *
 * tupla 0: possui
 *   i = (4 bytes) 1
 *   t = (11 bytes) 'joe's place'
 *   b = (5 bytes) \\000\\001\\002\\003\\004
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include "libpq-fe.h"

/* for ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

```

```

}

int
main(int argc, char **argv)
{
    const char    *conninfo;
    PGconn        *conn;
    PGresult       *res;
    const char    *paramValues[1];
    int           i,
                 j;
    int           i_fnum,
                 t_fnum,
                 b_fnum;

    /*
     * Se o usuário fornecer o parâmetro na linha de comando, este é
     * utilizado na cadeia de caracteres conninfo; senão, o padrão é
     * definir dbname=templatel e utilizar as variáveis de ambiente,
     * ou o valor padrão, para todos os outros parâmetros de conexão.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = templatel";

    /* Make a connection to the database */
    conn = PQconnectdb(conninfo);

    /* Check to see that the backend connection was successfully made */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /*
     * O objetivo deste programa é mostrar a utilização de PQexecParams()
     * com parâmetros fora-de-linha, assim como a transmissão binária dos
     * resultados. Utilizando parâmetros fora-de-linha pode-se evitar o
     * trabalho entediante de colocar apóstrofes e escapes. Deve ser
     * observado que não é necessário fazer nada especial com o apóstrofo
     * presente no valor do parâmetro.
     */

    /* Abaixo está o valor do parâmetro fora-de-linha */
    paramValues[0] = "joe's place";

    res = PQexecParams(conn,
                       "SELECT * FROM test1 WHERE t = $1",
                       1,                /* um parâmetro */
                       NULL,              /* o servidor deduz o tipo do parâmetro */
                       paramValues,
                       NULL,              /* não é necessário o comprimento do
                                           parâmetro, porque é texto */
                       NULL,              /* o padrão é todos os parâmetros no
                                           formato texto */
                       1);                /* solicitar resultados binários */

    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {

```

```

    fprintf(stderr, "SELECT falhou: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/* Utilizar PQfnumber para evitar assumir a ordem dos campos no resultado */
i_fnum = PQfnumber(res, "i");
t_fnum = PQfnumber(res, "t");
b_fnum = PQfnumber(res, "b");

for (i = 0; i < PQntuples(res); i++)
{
    char    *iptr;
    char    *tptr;
    char    *bptr;
    int     blen;
    int     ival;

    /* Obter os valores dos campos (ignorada a possibilidade de serem nulos) */
    iptr = PQgetvalue(res, i, i_fnum);
    tptr = PQgetvalue(res, i, t_fnum);
    bptr = PQgetvalue(res, i, b_fnum);

    /*
     * A representação de INT4 está na ordem de bytes da rede,
     * sendo melhor transformar para a ordem de bytes local.
     */
    ival = ntohl(*(uint32_t *) iptr);

    /*
     * A representação binária de TEXT é texto, e como a libpq anexa
     * um byte zero, funciona perfeitamente bem como uma cadeia de
     * caracteres C.
     *
     * A representação binária de BYTEA é um grupo de bytes, podendo
     * incluir nulos, portanto é necessário prestar atenção no
     * comprimento do campo.
     */
    blen = PQgetlength(res, i, b_fnum);

    printf("tupla %d: possui\n", i);
    printf(" i = (%d bytes) %d\n",
           PQgetlength(res, i, i_fnum), ival);
    printf(" t = (%d bytes) '%s'\n",
           PQgetlength(res, i, t_fnum), tptr);
    printf(" b = (%d bytes) ", blen);
    for (j = 0; j < blen; j++)
        printf("\\%03o", bptr[j]);
    printf("\n\n");
}

PQclear(res);

/* fechar a conexão com o banco de dados e limpar */
PQfinish(conn);

return 0;
}

```

Execução do programa exemplo: <sup>5</sup>

```
$ gcc testlibpq3.c -o testlibpq3 \
> -I /usr/local/pgsql/include/ \
> -L /usr/local/pgsql/lib/ -lpq
$ ./testlibpq3 "host=localhost user=teste password=teste dbname=teste"
```

```
tupla 0: possui
i = (4 bytes) 1
t = (11 bytes) 'joe's place'
b = (5 bytes) \000\001\002\003\004
```

## Notas

1. serviço — arquivo de configuração de conexão — Um serviço é um conjunto de parâmetros de conexão com nome. Podem ser especificados vários serviços no arquivo. Cada serviço começa pelo nome do serviço entre colchetes. As linhas seguintes contêm parâmetros de configuração de conexão com a forma "parâmetro=valor". As linhas que começam por '#' são comentários. (N. do T.)

ad hoc — para isso, para esse caso. Novo Dicionário Aurélio da Língua Portuguesa. (N. do T.)

3. Acréscimo feito pelo tradutor.
4. Acréscimo feito pelo tradutor.
5. Acréscimo feito pelo tradutor.

# Capítulo 28. Objetos grandes

O PostgreSQL possui a funcionalidade *objeto grande*, que fornece acesso na forma de fluxo aos dados dos usuários que são armazenados em uma estrutura especial de objeto grande. O acesso na forma de fluxo é útil quando se trabalha com valores de dados que são muito grandes para serem manuseados convenientemente como um todo.

Este capítulo descreve a implementação, e as interfaces de linguagem de programação e de consulta dos dados objeto grande no PostgreSQL. Nos exemplos deste capítulo é utilizada a biblioteca C `libpq`, mas a maioria das interfaces de programação nativas do PostgreSQL suportam funcionalidades equivalentes. Outras interfaces podem utilizar internamente a interface de objeto grande para fornecer suporte genérico a valores grandes, mas não são descritas aqui.

## 28.1. Histórico

O POSTGRES 4.2, predecessor indireto do PostgreSQL, suportava três implementações padrão para objetos grandes: como arquivos externos ao servidor POSTGRES; como arquivos externos gerenciados pelo servidor POSTGRES; e como dados armazenados dentro do banco de dados POSTGRES. Esta situação causava uma confusão considerável entre os usuários. Como consequência, somente permaneceu no PostgreSQL o suporte a objetos grandes como dados armazenados dentro do banco de dados. Embora seja mais lento para ser acessado, fornece uma integridade de dados mais rigorosa. Por motivos históricos, este esquema de armazenamento é referido como *Inversão de objetos grandes* (*Inversion large objects*) (Ocasionalmente será visto o termo *Inversão* utilizado com o mesmo significado de objeto grande). Desde o PostgreSQL 7.1 todos os objetos grandes são armazenados em uma tabela do sistema chamada `pg_largeobject`.

O PostgreSQL 7.1 introduziu o mecanismo apelidado de “TOAST” (fatias), que permite os valores dos dados serem muito maiores que as páginas de dados. Isto tornou a funcionalidade de objeto grande parcialmente obsoleta. Uma vantagem da funcionalidade de objeto grande que permaneceu, é permitir valores com tamanho de até 2 GB, enquanto os campos fatiados (`TOASTed`) podem ter no máximo 1 GB. Além disso, os objetos grandes podem ser manipulados pedaço a pedaço de maneira muito mais fácil que os campos de dados comuns e, portanto, os limites práticos são consideravelmente diferentes.

## 28.2. Funcionalidades da implementação

A implementação de objeto grande divide os objetos grandes em “pedaços” (`chunks`), e armazena estes pedaços em linhas no banco de dados. Um índice B-tree garante a procura rápida do número correto do pedaço quando são feitos acessos aleatórios de leitura e escrita.

## 28.3. Interfaces cliente

Esta seção descreve as facilidades que as bibliotecas de interface cliente do PostgreSQL fornecem para acessar objetos grandes. Toda manipulação de objeto grande que utiliza estas funções *deve* acontecer dentro de um bloco de transação SQL (Este requisito é exigido desde o PostgreSQL 6.5, embora tenha sido um requisito implícito nas versões anteriores, resultando em um mal comportamento quando ignorado). A interface de objeto grande do PostgreSQL é modelada segundo a interface do sistema de arquivos do Unix, com funções `open`, `read`, `write`, `lseek`, etc. análogas.

Os aplicativos cliente que utilizam a interface de objeto grande da `libpq` devem incluir o arquivo de cabeçalho `libpq/libpq-fs.h`, e fazer a ligação com a biblioteca `libpq`.

### 28.3.1. Criação de objeto grande

A função

```
Oid lo_creat(PGconn *conn, int modo);
```

cria um objeto grande novo. O argumento *modo* é uma máscara de bits que descreve vários atributos diferentes do novo objeto. As constantes simbólicas usadas aqui são definidas no arquivo de cabeçalho `libpq/libpq-fs.h`. O tipo de acesso (leitura, escrita ou ambos) é controlado pelo OU lógico dos bits de `INV_READ` e `INV_WRITE`. Os dezesseis bits de mais baixa ordem da máscara têm sido utilizados em Berkeley, historicamente, para designar o número do gerenciador de armazenamento no qual o objeto grande deve residir. Agora estes bits devem ser sempre zero. O valor retornado é o OID

atribuído ao novo objeto grande (o tipo de acesso também não faz mais nada, mas deve ser definido pelo menos um dos sinalizadores para evitar erro). O valor retornado é o OID atribuído ao novo objeto grande, ou InvalidOid (zero) se não for bem-sucedido.

Exemplo:

```
inv_oid = lo_creat(conn, INV_READ|INV_WRITE);
```

### 28.3.2. Importação de objeto grande

Para importar um arquivo do sistema operacional como um objeto grande é chamada a função:

```
Oid lo_import(PGconn *conn, const char *nome_do_arquivo);
```

O argumento *nome\_do\_arquivo* especifica o nome do arquivo do sistema operacional a ser importado para o novo objeto grande. O valor retornado é o OID atribuído ao novo objeto grande, ou InvalidOid (zero) se não for bem-sucedido. Deve ser observado que o arquivo é lido pela biblioteca de interface cliente, e não pelo servidor; portanto, o arquivo deve residir no sistema de arquivos do cliente e poder ser lido pelo aplicativo cliente.

### 28.3.3. Exportação de objeto grande

Para exportar um objeto grande para um arquivo do sistema operacional é chamada a função:

```
int lo_export(PGconn *conn, Oid lobjId, const char *nome_do_arquivo);
```

O argumento *lobjId* especifica o OID do objeto grande a ser exportado, e o argumento *nome\_do\_arquivo* especifica o nome do arquivo no sistema operacional. Deve ser observado que o arquivo é escrito pela biblioteca de interface cliente, e não pelo servidor. A função retorna 1 quando é bem-sucedida, ou -1 caso contrário.

### 28.3.4. Abertura objeto grande existente

Para abrir um objeto grande existente para ler ou escrever chama-se a função:

```
int lo_open(PGconn *conn, Oid lobjId, int modo);
```

O argumento *lobjId* especifica o OID do objeto grande a ser aberto. Os bits de *modo* controlam se o objeto deve ser aberto para leitura (INV\_READ), escrita (INV\_WRITE), ou ambos. O objeto grande não pode ser aberto antes de ser criado. A função *lo\_open* retorna o descritor do objeto grande (não negativo) para uso posterior em *lo\_read*, *lo\_write*, *lo\_lseek*, *lo\_tell* e *lo\_close*. O descritor é válido apenas pela duração da transação corrente. Quando a função não é bem-sucedida retorna -1.

### 28.3.5. Escrita de dados em objeto grande

A função

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

*writes* escreve *len* bytes de *buf* no descritor de objeto grande *fd*. O argumento *fd* deve ter sido retornado por uma chamada anterior a *lo\_open*. A função retorna o número de bytes realmente escritos. Caso aconteça algum erro, retorna um valor negativo.

### 28.3.6. Leitura de dados de objeto grande

A função

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

lê *len* bytes do descritor de objeto grande *fd* colocando-os em *buf*. O argumento *fd* deve ter sido retornado por uma chamada anterior à função *lo\_open*. A função retorna o número de bytes realmente lidos. Caso aconteça algum erro, retorna um valor negativo.

### 28.3.7. Procura em objeto grande

Para mudar a posição corrente de leitura ou de escrita associada ao descritor do objeto grande chama-se a função:

```
int lo_lseek(PGconn *conn, int fd, int deslocamento, int donde);
```

Esta função move o ponteiro de posição corrente do descritor de objeto grande, identificado por `fd`, para a nova posição especificada pelo argumento `deslocamento`. Os valores válidos para o argumento `donde` são `SEEK_SET` (procurar a partir do início do objeto), `SEEK_CUR` (procurar a partir da posição corrente), e `SEEK_END` (procurar a partir do fim do objeto). O valor retornado é o novo ponteiro de posição, ou -1 se não for bem-sucedida.

### 28.3.8. Obtenção da posição de procura no objeto grande

Para obter a posição corrente de leitura ou escrita do descritor de objeto grande chama-se a função:

```
int lo_tell(PGconn *conn, int fd);
```

No caso de erro retorna um valor negativo.

### 28.3.9. Fechamento do descritor do objeto grande

O descritor de objeto grande pode ser fechado chamando a função

```
int lo_close(PGconn *conn, int fd);
```

onde o argumento `fd` é o descritor do objeto grande retornado pela função `lo_open`. Se for bem-sucedida, a função `lo_close` retorna zero. Se houver erro, retorna um valor negativo.

Todo descritor de objeto grande que permanecer aberto no final da transação será fechado automaticamente.

### 28.3.10. Remoção de objeto grande

Para remover um objeto do grande do banco de dados chama-se a função:

```
int lo_unlink(PGconn *conn, Oid lobjId);
```

O argumento `lobjId` especifica o OID do objeto grande a ser removido. A função retorna 1 quando é bem-sucedida. No caso de erro retorna -1.

## 28.4. Funções do lado servidor

Existem duas funções do lado servidor, que podem ser chamadas através da linguagem SQL, que correspondem às duas funções do lado cliente descritas acima; na verdade, a maior parte das funções do lado cliente são simplesmente interfaces para funções equivalentes do lado servidor. As funções realmente úteis para serem chamadas através de comandos SQL são `lo_creat`, `lo_unlink`, `lo_import` e `lo_export`. Abaixo seguem exemplos de como utilizá-las:

```
CREATE TABLE imagem (
    nome          text,
    raster        oid
);

SELECT lo_creat(-1);          -- retorna o OID do objeto grande novo e vazio

SELECT lo_unlink(173454);    -- apaga o objeto grande com OID igual a 173454

INSERT INTO imagem (nome, raster)
VALUES ('uma linda imagem', lo_import('/etc/motd'));

SELECT lo_export(imagem.raster, '/tmp/motd') FROM imagem
WHERE nome = 'uma linda imagem';
```



As funções do lado servidor `lo_import` e `lo_export` se comportam de maneira consideravelmente diferente das suas funções análogas do lado cliente. Estas duas funções lêem e escrevem arquivos no sistema de arquivos do servidor, usando as permissões do usuário sob o qual o banco de dados executa. Portanto, o uso destas funções é restrito aos superusuários. Em contraposição, as funções de importação e exportação do lado cliente lêem e escrevem arquivos no sistema de arquivos do cliente, usando as permissões do programa cliente. As funções do lado cliente podem ser utilizadas por qualquer usuário do PostgreSQL.

## 28.5. Programa exemplo

O Exemplo 28-1 é um programa modelo, que mostra como a interface de objeto grande da biblioteca `libpq` pode ser utilizada. Partes do programa foram transformadas em comentário, mas foram deixadas no código fonte para benefício do leitor. Este programa também pode ser encontrado em `src/test/examples/testlo.c` na distribuição do código fonte.

### Exemplo 28-1. Programa de exemplo de objeto grande com `libpq`

```
/*-----
 *
 * testlo.c
 * teste de utilização de objetos grandes com libpq
 *
 * Portions Copyright (c) 1996-2005, PostgreSQL Global Development Group
 * Portions Copyright (c) 1994, Regents of the University of California
 *
 *
 * IDENTIFICATION
 * $PostgreSQL: postgresql/src/test/examples/testlo.c,v 1.25 2004/12/31 22:03:58 postgres Exp $
 *-----
 */
#include <stdio.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE 1024

/*
 * importFile -
 * importar o arquivo "in_filename" para o banco de dados
 * como o objeto grande "lobjOid"
 */
static Oid
importFile(PGconn *conn, char *filename)
{
    Oid    lobjId;
    int    lobj_fd;
    char   buf[BUFSIZE];
    int    nbytes,
           tmp;
    int    fd;

    /*
     * abrir o arquivo a ser lido
     */
}
```

```

fd = open(filename, O_RDONLY, 0666);
if (fd < 0)
{
    /* erro */
    fprintf(stderr,
        "não foi possível abrir o arquivo Unix\"%s\"\\n",
        filename);
}

/*
 * criar o objeto grande
 */
lobjId = lo_creat(conn, INV_READ | INV_WRITE);
if (lobjId == 0)
    fprintf(stderr, "não foi possível criar o objeto grande");

lobj_fd = lo_open(conn, lobjId, INV_WRITE);

/*
 * ler do arquivo Unix e escrever no arquivo de inversão
 */
while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
{
    tmp = lo_write(conn, lobj_fd, buf, nbytes);
    if (tmp < nbytes)
        fprintf(stderr,
            "erro durante a leitura de \"%s\"\\n",
            filename);
}

close(fd);
lo_close(conn, lobj_fd);

return lobjId;
}

static void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int    lobj_fd;
    char   *buf;
    int    nbytes;
    int    nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr,
            "não foi possível abrir o objeto grande %u",
            lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = '\\0';
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
        if (nbytes <= 0)
            break;
        /* sem mais dados? */
    }
}

```

```

    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

static void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int    lobj_fd;
    char   *buf;
    int     nbytes;
    int     nwritten;
    int     i;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "não foi possível abrir o objeto grande %u", lobjId);
    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);
    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = '\0';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
        if (nbytes <= 0)
        {
            fprintf(stderr, "\nERRO DE ESCRITA!\n");
            break;
        }
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

/*
 * exportFile -
 *     exportar o objeto grande "lobjOid" para o arquivo "out_filename"
 *
 */
static void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int    lobj_fd;
    char   buf[BUFSIZE];
    int     nbytes,
            tmp;
    int     fd;

    /*
     * criar um "objeto" inversão
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "não foi possível abrir o objeto grande %u", lobjId);
    /*
     * abrir o arquivo a ser escrito

```

```

    */
    fd = open(filename, O_CREAT | O_WRONLY | O_TRUNC, 0666);
    if (fd < 0)
    {
        fprintf(stderr,
            "não foi possível abrir o arquivo Unix \"%s\"",
            filename);
    }
    /*
    * ler do arquivo Unix e escrever no arquivo inversão
    */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr,
                "erro ao escrever \"%s\"",
                filename);
        }
    }

    lo_close(conn, lobj_fd);
    close(fd);

    return;
}

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char        *in_filename,
               *out_filename;
    char        *database;
    Oid         lobjOid;
    PGconn      *conn;
    PGresult     *res;

    if (argc != 4)
    {
        fprintf(stderr, "Utilização: %s nome_do_banco_de_dados \
nome_do_arquivo_de_entrada nome_do_arquivo_de_saída\n",
            argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
    * estabelecer a conexão
    */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

```

```

/* verificar se a conexão com o servidor foi bem-sucedida */
if (PQstatus(conn) != CONNECTION_OK)
{
    fprintf(stderr,
            "A conexão com o banco de dados %s falhou",
            PQerrorMessage(conn));
    exit_nicely(conn);
}

res = PQexec(conn, "begin");
PQclear(res);
printf("importing file \"%s\" ...\n", in_filename);
/* lobjOid = importFile(conn, in_filename); */
lobjOid = lo_import(conn, in_filename);
if (lobjOid == 0)
    fprintf(stderr, "%s\n", PQerrorMessage(conn));
else
{
    printf("\tcomo o objeto grande %u.\n", lobjOid);

    printf("lendo os bytes 1000-2000 do objeto grande\n");
    pickout(conn, lobjOid, 1000, 1000);

    printf("sobrescrevendo os bytes 1000-2000 do objeto grande com X's\n");
    overwrite(conn, lobjOid, 1000, 1000);

    printf("exportando o objeto grande para o arquivo \"%s\" ...\n",
           out_filename);
/* exportFile(conn, lobjOid, out_filename); */
    if (!lo_export(conn, lobjOid, out_filename))
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
}

res = PQexec(conn, "end");
PQclear(res);
PQfinish(conn);
return 0;
}

```

## Capítulo 29. ECPG - SQL incorporado à linguagem C

Este capítulo descreve o pacote de SQL incorporado para o PostgreSQL. Foi escrito por Linus Tolke (<linus@epact.se>) e Michael Meskes (<meskes@postgresql.org>). Originalmente escrito para trabalhar com a linguagem C, também trabalha com a linguagem C++, mas ainda não reconhece todas as construções de C++.

Esta documentação é bastante incompleta, mas uma vez que a interface é padronizada, podem ser encontradas informações adicionais em várias fontes sobre SQL.

### 29.1. O conceito

Um programa com SQL incorporado consiste de código escrito em uma linguagem de programação comum, neste caso a linguagem C, misturado com comandos SQL em seções com marcas especiais. Para construir o programa, primeiro o código fonte é passado através de um pré-processador de SQL incorporado, que converte este código em um programa C comum. Após isto ser feito, o código pode ser processado pelo compilador C.

O SQL incorporado possui vantagens sobre outros métodos que tratam comandos SQL a partir do código C: Em primeiro lugar, toma conta da entediante passagem de informação de e para as variáveis do programa escrito em C; Em segundo lugar, o código SQL incorporado ao programa é verificado quanto à correção sintática em tempo de construção; Em terceiro lugar, o SQL incorporado à linguagem C é especificado pelo padrão SQL, e suportado por muitos outros sistemas gerenciadores de banco de dados SQL.

A implementação do PostgreSQL foi projetada para corresponder ao padrão tanto quanto o possível. Geralmente é possível portar para o PostgreSQL, com relativa facilidade, programas com SQL incorporado escritos para outros gerenciadores de banco de dados SQL.

Como já foi dito, os programas escritos para a interface de SQL incorporado são programas C normais, com código especial inserido para realizar ações relacionadas com o banco de dados. Este código especial sempre possui a forma:

```
EXEC SQL ...;
```

Sintaticamente estas declarações tomam o lugar da declaração C. Dependendo da declaração em particular, pode aparecer no nível global ou dentro de uma função. As declarações SQL incorporadas seguem as regras de tratamento de letras maiúsculas e minúsculas do código SQL normal, e não as regras da linguagem C.

As seções que se seguem explicam todas as declarações SQL incorporadas.

### 29.2. Conexão com o servidor de banco de dados

A conexão com o servidor de banco de dados é feita utilizando a seguinte declaração:

```
EXEC SQL CONNECT TO destino [AS nome_da_conexão] [USER nome_do_usuario];
```

O *destino* pode ser especificado das seguintes formas:

- *nome\_do\_banco\_de\_dados*[@*nome\_do\_hospedeiro*][:*porta*]
- tcp:postgresql://*nome\_do\_hospedeiro*[:*porta*][/*nome\_do\_banco\_de\_dados*][?*opções*]
- unix:postgresql://*nome\_do\_hospedeiro*[:*porta*][/*nome\_do\_banco\_de\_dados*][?*opções*]
- literal cadeia de caracteres SQL contendo uma das formas acima
- referência a uma variável do tipo caractere contendo uma das formas acima (veja os exemplos)
- DEFAULT

Se o destino da conexão for especificado literalmente (ou seja, não for especificado através de referência a uma variável), e o valor não estiver entre aspas, então são aplicadas as regras normais do SQL para tratamento de letras maiúsculas e minúsculas. Neste caso, os parâmetros podem ser colocados individualmente entre aspas conforme seja necessário. Na prática, provavelmente há menos chance de errar quando se usa literais cadeia de caracteres (entre apóstrofes), ou

referência a uma variável. O destino de conexão `DEFAULT` inicia uma conexão com o banco de dados padrão sob o nome de usuário padrão. Neste caso, não é necessário especificar o nome do usuário ou da conexão em separado.

Também existem maneiras diferentes de especificar o nome do usuário:

- `nome_do_usuario`
- `nome_do_usuario/senha`
- `nome_do_usuario IDENTIFIED BY senha`
- `nome_do_usuario USING senha`

Como acima, os parâmetros `nome_do_usuario` e `senha` podem ser um identificador SQL, um literal cadeia de caracteres SQL, ou referência a uma variável do tipo caractere.

O `nome_da_conexao` é utilizado para tratar várias conexões em um mesmo programa. Pode ser omitido se o programa utilizar apenas uma conexão. A conexão aberta mais recentemente se torna a conexão corrente, que é utilizada por padrão quando uma declaração SQL é executada (veja mais adiante neste capítulo).

Abaixo estão mostrados alguns exemplos de declaração `CONNECT`:

```
EXEC SQL CONNECT TO meubanco@sql.meudominio.com;
```

```
EXEC SQL CONNECT TO 'unix:postgresql://sql.meudominio.com/meubanco' AS minhaconexao USER joao;
```

```
EXEC SQL BEGIN DECLARE SECTION;
const char *destino = "meubanco@sql.meudominio.com";
const char *usuario = "joao";
EXEC SQL END DECLARE SECTION;
...
EXEC SQL CONNECT TO :destino USER :usuario;
```

A última forma faz uso da variante que foi referida acima como referência a uma variável do tipo caractere. Na Seção 29.6 é mostrado como as variáveis da linguagem C podem ser utilizadas nas declarações SQL quando são prefixadas por dois-pontos (:).

Como o formato do destino da conexão não é especificado pelo padrão SQL, se for desejado desenvolver aplicativos portáteis deve-se dar preferência à utilização de algo baseado no último exemplo acima, para encapsular a cadeia de caracteres de destino da conexão fora da declaração.

## 29.3. Fechamento de conexão

Para fechar a conexão deve ser utilizada a seguinte declaração:

```
EXEC SQL DISCONNECT [conexão];
```

Onde `conexão` pode ser especificada das seguintes maneiras:

- `nome_da_conexao`
- `DEFAULT`
- `CURRENT`
- `ALL`

Se não for especificado o nome da conexão, a conexão corrente é fechada.

É bom estilo o aplicativo sempre fechar todas as conexões abertas.

## 29.4. Execução de comandos SQL

Pode ser executado qualquer comando SQL de dentro de um aplicativo com SQL incorporado. Abaixo estão mostrados alguns exemplos de como se faz isto.

Criação de tabela:

```
EXEC SQL CREATE TABLE foo (numero integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(numero);
EXEC SQL COMMIT;
```

Inserção de linhas:

```
EXEC SQL INSERT INTO foo (numero, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

Exclusão de linhas:

```
EXEC SQL DELETE FROM foo WHERE numero = 9999;
EXEC SQL COMMIT;
```

Seleção de uma única linha:

```
EXEC SQL SELECT numero INTO :numero FROM foo WHERE ascii = 'doodad';
```

Seleção usando cursores:

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT numero, ascii FROM foo
    ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :numero, :ascii;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

Atualizações:

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE numero = 9999;
EXEC SQL COMMIT;
```

Os termos (tokens) com a forma *:alguma\_coisa* são *variáveis hospedeiras*, ou seja, se referem a variáveis do programa C, e são explicadas na Seção 29.6.

No modo padrão, as declarações são efetivadas apenas quando é executado `EXEC SQL COMMIT`. A interface de SQL incorporado também suporta a auto-efetivação das transações (semelhante ao comportamento da biblioteca libpq), através da opção de linha de comando `-t` do `ecpg` (veja abaixo), ou através da declaração `EXEC SQL SET AUTOCOMMIT TO ON`. No modo de auto-efetivação todo comando é efetivado automaticamente, a menos que esteja dentro de um bloco de transação explícito. Este modo pode ser desabilitado explicitamente através da declaração `EXEC SQL SET AUTOCOMMIT TO OFF`.

## 29.5. Escolha da conexão

As declarações SQL mostradas na seção anterior são executadas na conexão corrente, ou seja, a conexão aberta mais recentemente. Se o aplicativo precisar gerenciar várias conexões, então existem duas maneiras de tratar este problema.

A primeira opção é escolher explicitamente a conexão para cada declaração SQL como, por exemplo:

```
EXEC SQL AT nome_da_conexão SELECT ...;
```

Esta opção é particularmente apropriada quando o aplicativo necessita utilizar várias conexões sem uma ordem definida.

Se o aplicativo utilizar vários fluxos de execução (threads), estes não podem compartilhar simultaneamente uma conexão. Deve ser controlado explicitamente o acesso às conexões (utilizando mutexes), ou utilizada uma conexão para cada fluxo de execução. Se cada fluxo de execução utilizar sua própria conexão, será necessário usar a cláusula `AT` para especificar a conexão que o fluxo de execução vai utilizar.

A segunda opção é executar uma declaração para alternar a conexão corrente. A declaração é:

```
EXEC SQL SET CONNECTION nome_da_conexão;
```



Esta opção é particularmente apropriada quando são executadas muitas declarações utilizando a mesma conexão, mas não considera os fluxos de execução.

## 29.6. Utilização de variáveis hospedeiras

Na Seção 29.4 foi visto como executar declarações SQL a partir de um programa com SQL incorporado. Algumas destas declarações utilizam apenas valores fixos, não possuindo uma maneira de inserir valores fornecidos pelo usuário na declaração, nem os programas processam os valores retornados pelas consultas. Este tipo de declaração, na verdade, não é útil em aplicativos reais. Esta seção explica em detalhe como passar dados entre o programa C e os comandos SQL incorporados, utilizando um mecanismo simples chamado *variáveis hospedeiras*.

### 29.6.1. Visão geral

A passagem de dados entre o programa C e as declarações SQL é particularmente simples no SQL incorporado. Em vez do programa colar os dados na declaração, que envolve várias dificuldades, como colocar o valor entre apóstrofes de forma apropriada, pode-se simplesmente escrever o nome da variável C, prefixada por dois-pontos, na declaração SQL. Por exemplo:

```
EXEC SQL INSERT INTO alguma_tabela VALUES (:v1, 'foo', :v2);
```

Esta declaração faz referência a duas variáveis C, chamadas `v1` e `v2`, e também usa um literal cadeia de caracteres SQL regular, para mostrar que não fica restrito ao uso de um tipo de dado ou outro.

Esta forma de inserir variáveis C em declarações SQL funciona em qualquer lugar onde é esperada uma expressão de valor na declaração SQL. No ambiente SQL, as variáveis C referenciadas são chamadas de *variáveis hospedeiras*.

### 29.6.2. Seções de declaração

Para passar os dados do programa para o banco de dados, por exemplo como parâmetros de uma consulta, ou para passar os dados do banco de dados de volta para o programa C, as variáveis C utilizadas para armazenar estes dados precisam ser declaradas em seções marcadas de forma especial, para que o pré-processador de SQL incorporado tenha conhecimento destas variáveis.

Estas seções começam por

```
EXEC SQL BEGIN DECLARE SECTION;
```

e terminam por

```
EXEC SQL END DECLARE SECTION;
```

Entre estas duas linhas devem haver declarações normais de variáveis C, como:

```
int    x;
char  foo[16], bar[16];
```

Podem existir tantas seções de declaração no programas quantas forem desejadas.

As declarações também são reproduzidas no arquivo de saída como variáveis C normais, para que não seja necessário declará-las novamente. As variáveis que não são utilizadas nos comandos SQL podem ser declaradas normalmente fora destas seções especiais.

A definição das estruturas e das uniões também devem ser colocadas dentro da seção `DECLARE`, senão o pré-processador não pode tratar estes tipos, uma vez que não conhece as definições.

O tipo especial `VARCHAR` é convertido em uma `struct` com nome para todas as variáveis. Uma declaração como

```
VARCHAR var[180];
```

é convertida em

```
struct varchar_var { int len; char arr[180]; } var;
```

Esta estrutura é adequada para servir de interface para os dados do tipo `varchar` do SQL.<sup>1</sup>

### 29.6.3. Declarações `SELECT INTO` e `FETCH INTO`

Agora já se sabe como passar dados gerados pelo programa para o comando SQL, mas como fazer para trazer os resultados da consulta? Para esta finalidade o SQL incorporado disponibiliza duas variantes dos comandos usuais `SELECT` e `FETCH`. Estes comandos possuem uma cláusula especial `INTO`, que especifica em quais variáveis hospedeiras os valores trazidos são armazenados.

Abaixo segue um exemplo:

```
/*
 * assumindo a existência desta tabela:
 * CREATE TABLE test1 (a int, b varchar(50));
 */

EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test1;
```

Portanto, a cláusula `INTO` aparece entre a lista de seleção e a cláusula `FROM`. O número de elementos da lista de seleção e da lista após a cláusula `INTO`, também chamada de lista de destino, devem ser iguais.

Abaixo segue um exemplo mostrando o comando `FETCH`:

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test1;

...

do {
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
    ...
} while (...);
```

Aqui a cláusula `INTO` aparece após todas as cláusulas normais.

Estes dois métodos só permitem trazer uma linha de cada vez. Se for necessário processar conjunto de resultados contendo potencialmente mais de uma linha, é necessário utilizar um cursor, conforme mostrado no segundo exemplo.

### 29.6.4. Indicadores

Os exemplos acima não tratam valores nulos. Na verdade, os exemplos de recuperação vão lançar um erro se trouxerem um valor nulo do banco de dados. Para ser possível passar valores nulos para o banco de dados, ou trazer valores nulos do banco de dados, é necessário anexar uma segunda variável hospedeira na especificação de cada variável hospedeira que contém dado. Esta segunda variável é chamada de *indicador*, e contém um sinalizador para informar se o dado é nulo. Neste caso o valor da variável hospedeira real é ignorado. Abaixo está mostrado um exemplo que trata a recuperação de valores nulos de forma correta:

```
EXEC SQL BEGIN DECLARE SECTION;
int v1, ind1, ind2;
```

```

VARCHAR v2;
EXEC SQL END DECLARE SECTION;

...

EXEC SQL SELECT a, b INTO :v1 :ind1, :v2 :ind2 FROM test1;

```

A variável indicadora (`ind1` ou `ind2`) será igual a zero quando o valor da coluna não for nulo, ou será negativa quando o valor da coluna for nulo. Deve ser observado que não há vírgula separando a variável hospedeira da variável indicadora. Não há necessidade de espaço entre a variável hospedeira e a variável indicadora.

O indicador possui outra função: se o valor do indicador for positivo, significa que o valor não é nulo, mas que foi truncado ao ser armazenado na variável hospedeira.<sup>2 3</sup>

## 29.7. SQL dinâmico

Em muitos casos, a declaração SQL a ser utilizada pelo aplicativo já é conhecida na hora em que o aplicativo é escrito. Em alguns casos, entretanto, as declarações SQL são formadas em tempo de execução, ou fornecidas por uma fonte externa. Nestes casos, a declaração SQL não pode ser incorporada diretamente ao código fonte C, mas existe um mecanismo que permite executar declarações SQL arbitrárias especificadas através de variáveis do tipo cadeia de caractere.

A forma mais simples de executar uma declaração SQL arbitrária é utilizando o comando `EXECUTE IMMEDIATE`. Por exemplo:

```

EXEC SQL BEGIN DECLARE SECTION;
const char *declaracao = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :declaracao;

```

As declarações que trazem dados (por exemplo, `SELECT`), não podem ser executadas desta forma.

Uma forma mais poderosa de executar declarações SQL arbitrárias é preparar uma vez, e executar a declaração preparada tantas vezes quanto se desejar. Também é possível preparar uma versão generalizada da declaração, e executar versões específicas desta fazendo a substituição de parâmetros. Ao preparar a declaração são colocados pontos de interrogação nos locais a serem substituídos posteriormente por parâmetros. Por exemplo:

```

EXEC SQL BEGIN DECLARE SECTION;
const char *declaracao = "INSERT INTO test1 VALUES(?, ?)";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE minha_declaracao FROM :declaracao;
...
EXEC SQL EXECUTE minha_declaracao USING 42, 'foobar';

```

Quando a declaração utilizada retorna valores é adicionada a cláusula `INTO`:

```

EXEC SQL BEGIN DECLARE SECTION;
const char *declaracao = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3;
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE minha_declaracao FROM :declaracao;
...
EXEC SQL EXECUTE minha_declaracao INTO v1, v2, v3 USING 37;

```

O comando `EXECUTE` pode possuir uma cláusula `INTO`, uma cláusula `USING`, as duas, ou nenhuma delas.

Quando a declaração preparada não é mais necessária deve-se liberá-la:

```

EXEC SQL DEALLOCATE PREPARE nome;

```

## 29.8. Utilização das áreas descritoras de SQL

A área descritora de SQL é um método mais sofisticado para processar os resultados das declarações `SELECT` e `FETCH`. A área descritora de SQL agrupa os dados de uma linha de dados junto com os itens de metadado, em uma estrutura de dados. Os metadados são particularmente úteis ao executar declarações SQL dinâmicas, quando a natureza das colunas do resultado não podem ser conhecidas a priori.

Uma área descritora de SQL consiste em um cabeçalho, que contém informações a respeito de todo o descritor, e uma ou mais áreas descritoras de item, onde cada área basicamente descreve uma coluna da linha de resultado.

É necessário alocar a área descritora de SQL para que possa ser utilizada:

```
EXEC SQL ALLOCATE DESCRIPTOR identificador;
```

O identificador serve como o “nome de variável” da área descritora. Quando não há mais necessidade do descritor, este deve ser liberado:

```
EXEC SQL DEALLOCATE DESCRIPTOR identificador;
```

Para usar a área descritora deve-se especificá-la como destino do armazenamento na cláusula `INTO`, em vez de especificar as variáveis hospedeiras:

```
EXEC SQL FETCH NEXT FROM meu_cursor INTO DESCRIPTOR meu_descritor;
```

Como fazer para extrair os dados da área descritora? Pode-se pensar na área descritora como sendo uma estrutura com campos nomeados. Para extrair o valor de um campo do cabeçalho e armazená-lo em uma variável hospedeira deve ser utilizada a seguinte declaração:

```
EXEC SQL GET DESCRIPTOR nome :variável_hospedeira = campo;
```

Atualmente existe apenas um campo de cabeçalho definido, chamado `COUNT`, que informa quantas áreas descritoras de item existem (ou seja, quantas colunas o resultado contém). A variável hospedeira precisa ser do tipo inteiro. Para extrair um campo da área descritora de item deve ser utilizada a seguinte declaração:

```
EXEC SQL GET DESCRIPTOR nome VALUE num :variável_hospedeira = campo;
```

onde *num* pode ser um literal inteiro, ou uma variável hospedeira contendo um inteiro. Os campos possíveis são:

`CARDINALITY` (integer)

número de linhas no conjunto de resultados

`DATA`

item de dado verdadeiro (portanto, o tipo de dado deste campo depende da consulta)

`DATETIME_INTERVAL_CODE` (integer)

?

`DATETIME_INTERVAL_PRECISION` (integer)

não implementado

`INDICATOR` (integer)

o indicador (indica um valor nulo ou o truncamento do valor)

`KEY_MEMBER` (integer)

não implementado

`LENGTH` (integer)

comprimento do dado em caracteres

NAME (string)	nome da coluna
NULLABLE (integer)	não implementado
OCTET_LENGTH (integer)	comprimento da representação do caractere do dado em bytes
PRECISION (integer)	precisão (para o tipo <code>numeric</code> )
RETURNED_LENGTH (integer)	comprimento do dado em caracteres
RETURNED_OCTET_LENGTH (integer)	comprimento da representação do caractere do dado em bytes
SCALE (integer)	escala (para o tipo <code>numeric</code> )
TYPE (integer)	código numérico do tipo de dado da coluna

## 29.9. Tratamento de erro

Esta seção descreve como tratar condições excepcionais e advertências em programas com SQL incorporado. Existem várias maneiras não exclusivas para isto.

### 29.9.1. Definição de chamada

Um método simples para capturar os erros e advertências é definir uma ação específica a ser executada sempre que uma determinada condição ocorrer. Em geral:

```
EXEC SQL WHENEVER condição ação;
```

a *condição* pode ser uma das seguintes:

SQLERROR

A ação especificada é chamada sempre que ocorre um erro durante a execução da declaração SQL.

SQLWARNING

A ação especificada é chamada sempre que ocorre uma advertência durante a execução da declaração SQL.

NOT FOUND

A ação especificada é chamada sempre que a declaração SQL traz ou afeta zero linhas (Esta condição não é um erro, mas pode-se estar interessado em tratá-la de forma especial).

a *ação* pode ser uma das seguintes:

CONTINUE

Significa efetivamente que a condição é ignorada. É o padrão.

GOTO *rótulo*

GO TO *rótulo*

Desvia para o rótulo especificado (utilizando a declaração `goto` da linguagem C).

**SQLPRINT**

Envia uma mensagem para a saída de erro padrão. É útil em programas simples ou durante a prototipação. Não é possível configurar os detalhes da mensagem.

**STOP**

Chama `exit(1)`, que termina o programa.

**BREAK**

Executa a declaração `break` da linguagem C. Deve ser utilizada apenas em laços ou nas declarações `switch`.

**CALL nome (args)****DO nome (args)**

Chama a função C com os argumentos especificados.

O padrão SQL especifica apenas as ações `CONTINUE` e `GOTO` (e `GO TO`).

Abaixo está mostrado um exemplo do que pode-se desejar utilizar em um programa simples. Estas declarações fazem com que seja impressa uma mensagem quando ocorre uma advertência, e interrompe o programa quando acontece um erro.

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

A declaração `EXEC SQL WHENEVER` é uma diretiva para o pré-processador de SQL, e não uma declaração C. As ações de erro ou de advertência definidas são aplicadas a todas as declarações SQL incorporadas que aparecem abaixo do ponto onde o tratador é definido, a menos que seja definida uma ação diferente para a mesma condição entre a primeira declaração `EXEC SQL WHENEVER` e a declaração SQL causadora da condição, a despeito do fluxo de controle no programa C. Portanto, nenhum dos trechos de programa C abaixo produz o efeito desejado.

```
/*
 * ERRADO
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}

/*
 * ERRADO
 */
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

### 29.9.2. sqlca

Para permitir um tratamento de erro mais poderoso, a interface de SQL incorporado disponibiliza uma variável global com o nome de `sqlca` que possui a seguinte estrutura:

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[70];
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

Em programas com vários fluxos de execução (*multithreaded*), cada fluxo de execução obtém automaticamente sua própria cópia da variável `sqlca`. Funciona de forma semelhante à variável C global `errno`.

A variável `sqlca` cobre tanto advertências quanto erros. Se ocorrerem vários erros ou advertências durante a execução de uma declaração, então a `sqlca` somente vai conter informações sobre a última ocorrência.

Se não ocorrer nenhum erro na última declaração SQL executada, então `sqlca.sqlcode` será 0 e `sqlca.sqlstate` será "00000". Se ocorrer um erro ou advertência, então `sqlca.sqlcode` terá um valor negativo e `sqlca.sqlstate` será diferente de "00000". Um valor positivo em `sqlca.sqlcode` indica uma condição inofensiva, como a última consulta ter retornado zero linhas. `sqlcode` e `sqlstate` possuem dois esquemas de código de erro diferentes; os detalhes são mostrados abaixo.

Se a última declaração SQL for bem sucedida, então `sqlca.sqlerrd[1]` conterá o OID da linha processada, quando aplicável, e `sqlca.sqlerrd[2]` conterá o número de linhas processadas ou retornadas, se aplicável ao comando.

Em caso de erro ou de advertência, `sqlca.sqlerrm.sqlerrmc` contém uma cadeia de caracteres que descreve o erro. O campo `sqlca.sqlerrm.sqlerrml` contém o comprimento da mensagem de erro armazenada em `sqlca.sqlerrm.sqlerrmc` (o resultado de `strlen()`, sem interesse real para um programador C). Deve ser observado que algumas mensagens são muito longas para caber em uma matriz `sqlerrmc` de tamanho fixo, sendo truncadas.

Em caso de advertência `sqlca.sqlwarn[2]` é definida como `w` (Em todos os outros casos é definida com um valor diferente de `w`). Se `sqlca.sqlwarn[1]` for definida como `w`, então o valor foi truncado quando foi armazenado na variável hospedeira. `sqlca.sqlwarn[0]` é definida como `w` quando algum dos outros elementos é definido para indicar uma advertência.

Atualmente os campos `sqlcaid`, `sqlcab`, `sqlerrp`, e o restante dos elementos de `sqlerrd` e de `sqlwarn`, não contêm informações úteis.

A estrutura `sqlca` não é definida no padrão SQL, mas é implementada em vários outros sistemas gerenciadores de banco de dados SQL. As definições são semelhantes no núcleo, mas se for desejado escrever aplicativos portáteis então deve-se investigar as diferentes implementações cuidadosamente.

### 29.9.3. SQLSTATE versus SQLCODE

Os campos `sqlca.sqlstate` e `sqlca.sqlcode` são dois esquemas diferentes de código de erro. Ambos são especificados pelo padrão SQL, mas `SQLCODE` foi marcado em obsolescência (*deprecated*) na edição do padrão de 1992, e removido na edição de 1999. Portanto, encoraja-se fortemente que os novos aplicativos utilizem `SQLSTATE`.

`SQLSTATE` é uma matriz de cinco caracteres. Os cinco caracteres contêm dígitos ou letras maiúsculas que representam códigos de vários erros e de condições de advertência. `SQLSTATE` possui um esquema hierárquico: os dois primeiros caracteres indicam a classe geral da condição, e os três últimos caracteres indicam a subclasse da condição geral. O status de bem-sucedido é indicado pelo código 00000. A maior parte dos códigos de `SQLSTATE` são definidos pelo padrão SQL.

O servidor PostgreSQL suporta nativamente os códigos de erro `SQLSTATE`; portanto, pode ser obtido um alto grau de consistência utilizando este esquema de código de erro através de todos os aplicativos. Para obter informações adicionais deve ser consultado o Apêndice A.

O esquema de código de erro obsoleto `SQLCODE` é um inteiro simples. O valor 0 indica bem-sucedido, um valor positivo indica bem-sucedido com informação adicional, e um valor negativo indica um erro. O padrão SQL define apenas o valor positivo +100, indicando que o último comando retornou ou afetou zero linhas, e não especifica nenhum valor negativo. Portanto, este esquema pode obter apenas uma portabilidade pobre, e não possui uma atribuição de código hierárquica. Historicamente, o pré-processador de SQL incorporado do PostgreSQL tem atribuído alguns valores específicos de `SQLCODE` para seu uso, os quais estão listados abaixo junto com seus valores numéricos e nomes simbólicos. Deve ser lembrado que estes valores não são portáveis para outras implementações do padrão SQL. Para simplificar a migração dos aplicativos para o esquema `SQLSTATE`, também é mostrado o `SQLSTATE` correspondente. Entretanto, não existe nenhum mapeamento um-para-um ou um-para-muitos entre estes dois esquemas (na verdade, é muitos-para-muitos), portanto deve ser consultada a lista de `SQLSTATE` global no Apêndice A para cada caso.

Abaixo estão mostrados os valores de `SQLCODE` atribuídos:

-12 (`ECPG_OUT_OF_MEMORY`)

Indica que a memória virtual encontra-se exaurida. (`SQLSTATE YE001`)

-200 (`ECPG_UNSUPPORTED`)

Indica que o pré-processador gerou algo que a biblioteca não tem conhecimento. Talvez esteja sendo usada uma versão do pré-processador incompatível com a da biblioteca. (`SQLSTATE YE002`)

-201 (`ECPG_TOO_MANY_ARGUMENTS`)

Significa que o comando especificou mais variáveis hospedeiras do que o comando espera. (`SQLSTATE 07001` ou `07002`)

-202 (`ECPG_TOO_FEW_ARGUMENTS`)

Significa que o comando especificou menos variáveis hospedeiras do que o comando espera. (`SQLSTATE 07001` ou `07002`)

-203 (`ECPG_TOO_MANY_MATCHES`)

Significa que a consulta retornou várias linhas, mas a declaração somente está preparada para armazenar uma linha de resultado (por exemplo, porque as variáveis especificadas não são matrizes). (`SQLSTATE 21000`)

-204 (`ECPG_INT_FORMAT`)

A variável hospedeira é do tipo `int`, e o dado no banco de dados é de um tipo diferente e contém um valor que não pode ser interpretado como um `int`. A biblioteca utiliza `strtol()` para esta conversão. (`SQLSTATE 42804`)

-205 (`ECPG_UINT_FORMAT`)

A variável hospedeira é do tipo `unsigned int` e o dado no banco de dados é de um tipo diferente e contém um valor que não pode ser interpretado como um `unsigned int`. A biblioteca utiliza `strtoul()` para esta conversão. (`SQLSTATE 42804`)

-206 (`ECPG_FLOAT_FORMAT`)

A variável hospedeira é do tipo `float` e o dado no banco de dados é de um tipo diferente e contém um valor que não pode ser interpretado como um `float`. A biblioteca utiliza `strtod()` para esta conversão. (`SQLSTATE 42804`)

-207 (`ECPG_CONVERT_BOOL`)

Significa que a variável hospedeira é do tipo `bool` e o dado no banco de dados não é nem `'t'` nem `'f'`. (`SQLSTATE 42804`)

-208 (`ECPG_EMPTY`)

A declaração enviada para o servidor PostgreSQL estava vazia (Normalmente isto não pode acontecer em um programa com SQL incorporado, portanto pode indicar um erro interno). (`SQLSTATE YE002`)

-209 (`ECPG_MISSING_INDICATOR`)

Foi retornado um valor nulo e não foi especificada uma variável indicadora. (`SQLSTATE 22002`)



-210 (ECPG\_NO\_ARRAY)

Foi utilizada uma variável comum em um local que requer uma matriz. (SQLSTATE 42804)

-211 (ECPG\_DATA\_NOT\_ARRAY)

O banco de dados retornou uma variável comum em um local que requer um valor matricial. (SQLSTATE 42804)

-220 (ECPG\_NO\_CONN)

O programa tentou acessar uma conexão que não existe. (SQLSTATE 08003)

-221 (ECPG\_NOT\_CONN)

O programa tentou acessar uma conexão que existe, mas que não está aberta (Isto é um erro interno). (SQLSTATE YE002)

-230 (ECPG\_INVALID\_STMT)

A declaração que está se tentando utilizar não foi preparada. (SQLSTATE 26000)

-240 (ECPG\_UNKNOWN\_DESCRIPTOR)

Não foi encontrado o descritor especificado. A declaração que está se tentando utilizar não foi preparada. (SQLSTATE 33000)

-241 (ECPG\_INVALID\_DESCRIPTOR\_INDEX)

O índice descritor especificado está fora do intervalo. (SQLSTATE 07009)

-242 (ECPG\_UNKNOWN\_DESCRIPTOR\_ITEM)

Foi requisitado um item descritor inválido (Este é um erro interno). (SQLSTATE YE002)

-243 (ECPG\_VAR\_NOT\_NUMERIC)

Durante a execução da declaração dinâmica o banco de dados retornou um valor numérico e a variável hospedeira não é numérica. (SQLSTATE 07006)

-244 (ECPG\_VAR\_NOT\_CHAR)

Durante a execução da declaração dinâmica o banco de dados retornou um valor não numérico e a variável hospedeira é numérica. (SQLSTATE 07006)

-400 (ECPG\_PGSQL)

Algum erro causado pelo servidor PostgreSQL. A mensagem contém a mensagem de erro do servidor PostgreSQL.

-401 (ECPG\_TRANS)

O servidor PostgreSQL sinalizou que não pode iniciar, efetivar ou desfazer a transação. (SQLSTATE 08007)

-402 (ECPG\_CONNECT)

A tentativa de conexão com o banco de dados não foi bem-sucedida. (SQLSTATE 08001)

100 (ECPG\_NOT\_FOUND)

É uma condição inofensiva, indicando que o último comando trouxe ou processou zero linhas, ou que se chegou ao fim do cursor. (SQLSTATE 02000)

## 29.10. Inclusão de arquivos

Para incluir um arquivo externo no programa com SQL incorporado deve ser utilizada a declaração:

```
EXEC SQL INCLUDE nome_do_arquivo;
```

O pré-processador de SQL incorporado procura pelo arquivo chamado *nome\_do\_arquivo.h*, pré-processa este arquivo, e o inclui na saída C produzida. Portanto, as declarações SQL presentes no arquivo incluído são tratadas de forma correta.

Deve ser observado que isto *não* é o mesmo que

```
#include <nome_do_arquivo.h>
```

porque este arquivo não estará sujeito ao pré-processamento de comandos SQL. Como é natural, pode-se continuar utilizando a diretiva `#include` da linguagem C para incluir outros arquivos de cabeçalho.

**Nota:** É feita diferenciação entre letras maiúsculas e minúsculas no nome do arquivo, embora o restante do comando `EXEC SQL INCLUDE` obedeça as regras de maiúsculas e minúsculas do SQL.

## 29.11. Processamento dos programas com SQL incorporado

Agora que já se tem uma idéia de como se escreve programas C com SQL incorporado, deseja-se saber como fazer para compilar estes programas. Antes de ser compilado, o programa é submetido ao pré-processador C de SQL incorporado, que converte as declarações SQL utilizadas em chamadas a funções especiais. Após a compilação, deve ser feita a ligação com uma biblioteca especial que contém as funções necessárias. Estas funções pegam informações dos argumentos, executam o comando SQL utilizando a interface libpq, e colocam o resultado nos argumentos especificados para saída.

O programa pré-processador chama-se `ecpg`, estando incluído na instalação normal do PostgreSQL. Os programas com SQL incorporado recebem tipicamente a extensão `.pgc`. Se houver um arquivo de programa chamado `prog1.pgc` este pode ser pré-processado chamando simplesmente:

```
ecpg prog1.pgc
```

Este comando cria um arquivo chamado `prog1.c`. Se os arquivos de entrada não seguirem o padrão de nomes sugerido, o nome do arquivo de saída pode ser especificado explicitamente utilizando a opção `-o`.

O arquivo pré-processado pode ser compilado normalmente. Por exemplo:

```
cc -c prog1.c
```

Os arquivos fontes C incluem arquivos de cabeçalho da instalação do PostgreSQL. Portanto, se o PostgreSQL estiver instalado em um local que não é procurado por padrão, é necessário adicionar opções como `-I/usr/local/pgsql/include` à linha de comando de compilação.

Para ligar o programa com SQL incorporado é necessário incluir a biblioteca `libecpg`, como:

```
cc -o myprog prog1.o prog2.o ... -lecpg
```

Novamente, pode ser necessário adicionar uma opção do tipo `-L/usr/local/pgsql/lib` à linha de comando.

Se o processo de construção de um projeto grande for gerenciado pelo `make`, pode ser conveniente incluir as seguintes regras implícitas nos arquivos de construção:

```
ECPG = ecpg

%.c: %.pgc
    $(ECPG) $<
```

A sintaxe completa do comando `ecpg` está descrita na `ecpg`.

A biblioteca `ecpg` é segura quanto a `thread` se for construída utilizando a opção de linha de comando `--enable-thread-safety` no `configure` (Pode ser necessário utilizar outras opções de linha de comando para `thread` para compilar o código cliente).

## 29.12. Funções da biblioteca

A biblioteca `libecpg` contém principalmente funções “escondidas”, utilizadas para implementar as funcionalidades expressas pelos comandos SQL incorporados. Mas existem algumas funções úteis que podem ser chamadas diretamente. Deve ser observado que isto torna o código não portátil.

- `ECPGdebug(int on, FILE *stream)` ativa o registro de depuração se for chamada com o primeiro argumento diferente de zero. O registro de depuração é feito em `stream`. O registro contém todas as declarações SQL junto com todas as variáveis de entrada inseridas, e o resultado do servidor PostgreSQL. Pode ser muito útil ao se procurar por erros nas declarações SQL.

- `ECPGstatus(int número_da_linha, const char* nome_da_conexão)` retorna verdade se estiver conectado ao banco de dados e falso se não estiver. O `nome_da_conexão` pode ser igual a `NULL` se estiver sendo utilizada uma única conexão.

## 29.13. Internamente

Esta seção explica como o ECPG trabalha internamente. Às vezes estas informações podem ser úteis para ajudar os usuários a entender como se usa o ECPG.

As quatro primeiras linhas escritas pelo `ecpg` na saída são linhas fixas. Duas delas são comentários e duas são linhas de inclusão necessárias para interface com a biblioteca. Depois o pré-processador lê o arquivo e escreve a saída. Normalmente, simplesmente reproduz o que foi lido na saída.

Quando encontra uma declaração `EXEC SQL` faz uma intervenção e muda a declaração. O comando começa por `EXEC SQL` e termina por `;`. Tudo entre estas duas partes é tratado como sendo uma declaração SQL, e analisada para substituição de variável.

A substituição de variável ocorre quando o símbolo começa por dois-pontos (`:`). A variável com este nome é procurada entre as variáveis que foram previamente declaradas na seção `EXEC SQL DECLARE`.

A função mais importante da biblioteca é `ECPGdo`, que toma conta da execução da maioria dos comandos. Recebe um número variável de argumentos. Pode chegar facilmente a 50 argumentos, e se espera que isto não seja um problema em nenhuma plataforma.

Os argumentos são:

Um número de linha

Este é o número de linha da linha original; utilizada apenas nas mensagens de erro.

Uma cadeia de caracteres

Este é o comando SQL a ser executado. É modificado pelas variáveis de entrada, ou seja, as variáveis que não eram conhecidas na hora da compilação, mas que devem ser introduzidas no comando. As posições onde as variáveis devem ser colocadas contêm `?`.

Variáveis de entrada

Cada variável de entrada causa a criação de 10 argumentos (veja abaixo).

`ECPGt_EOIT`

Um `enum` informando que não há mais variáveis de entrada.

Variáveis de saída

Cada variável de saída causa a criação de 10 argumento (veja abaixo). Estas variáveis são preenchidas pela função.

`ECPGt_EORT`

Um `enum` informando que não há mais variáveis.

Para cada variável que faz parte do comando SQL a função recebe 10 argumentos:

1. O tipo, como um símbolo especial.
2. Um ponteiro para o valor, ou um ponteiro para um ponteiro.
3. O tamanho da variável, se for do tipo `char` ou `varchar`.
4. O número de elementos da matriz (para trazer matrizes).
5. O deslocamento do próximo elemento da matriz (para trazer matrizes).
6. O tipo da variável indicadora, como um símbolo especial.
7. Um ponteiro para a variável indicadora.
8. 0
9. O número de elementos na matriz de indicadores (para trazer matrizes).
10. O deslocamento do próximo elemento na matriz de indicadores (para trazer matrizes).

Deve ser observado que nem todos os comandos SQL são tratados desta maneira. Por exemplo, uma declaração de abrir cursor como

```
EXEC SQL OPEN cursor;
```

não é copiada para a saída. Em vez disso, o comando DECLARE do cursor é utilizado na posição do comando OPEN, porque este realmente abre o cursor.

Abaixo está mostrado um exemplo completo que descreve a saída do pré-processador para o arquivo `foo.pgc` (podem haver detalhes diferentes entre versões diferentes do pré-processador):

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
int result;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :result FROM mytable WHERE index = :index;
```

é traduzido em:

```
/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql begin declare section */

#line 1 "foo.pgc"

    int index;
    int result;
/* exec sql end declare section */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM mytable WHERE index = ?      ",
        ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

(Neste caso foram adicionados recuos para ficar mais fácil de ler, mas não é algo que o pré-processador faça)

## 29.14. Exemplos

**Nota:** Seção escrita pelo tradutor, não fazendo parte do manual original.

Esta seção contém exemplos de programas com SQL incorporado à linguagem C. Para ser possível pré-compilar, compilar, gerar o executável e executar estes programas, foi feita a construção da árvore de fontes sob o diretório raiz do usuário (~), conforme mostrado abaixo. O arquivo de fontes `postgresql-8.0.0.tar.gz` foi baixado no diretório `/download`. Deve ser notado que não foi feita a instalação, somente foi feita a construção.

```
cd ~
tar xzvf /download/postgresql-8.0.0.tar.gz
cd postgresql-8.0.0/
./configure
make
All of PostgreSQL successfully made. Ready to install.
```

**Exemplo 29-1. Ler e mostrar o conteúdo de uma tabela**

Este exemplo mostra a utilização de comandos SQL incorporados ao fonte do programa C. O programa estabelece uma conexão com o servidor de banco de dados por meio de soquete do domínio Unix, acessa o banco de dados “teste” através do usuário cujo nome é “teste” e cuja senha também é “teste”. Depois lista as linhas da tabela criada pelo script mostrado abaixo:

```
CREATE TABLE pessoal (id SERIAL PRIMARY KEY, nome VARCHAR(50));
INSERT INTO pessoal (nome) VALUES ('Maria');
INSERT INTO pessoal (nome) VALUES ('Manuel');
INSERT INTO pessoal (nome) VALUES ('Francisco');
```

Abaixo está mostrado o código fonte do programa pessoal.pgc.

```
/*
 * Programa para mostrar a utilização do SQL estático incorporado.
 * Baseado em:
 *   The art of metaprogramming, Part 1: Introduction to metaprogramming
 *   http://www-128.ibm.com/developerworks/linux/library/l-metaprogl.html
 */

#include <stdio.h>

/*
 * Rotina para tratamento de erro
 */

static void erro() {
    printf("#%ld:%s\n", sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
    exit(1);
}

/*
 * Rotina para tratamento de advertência
 */

static void advertencia() {
    printf("#%ld:%s\n", sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
}

int main()
{
    /*
     * Tratar os erros e advertências de forma apropriada
     */

    EXEC SQL WHENEVER SQLWARNING DO advertencia(); /* sqlca.sqlwarn[0] == 'W' */
    EXEC SQL WHENEVER SQLERROR DO erro();          /* sqlca.sqlcode < 0 */

    /*
     * Estabelecer a conexão com o servidor de banco de dados.
     * Usar o nome, senha e banco de dados apropriados.
     */

    EXEC SQL CONNECT TO unix:postgresql://localhost/teste USER teste/teste;

    /*
     * Variáveis utilizadas para armazenamento
     * temporário dos campos do banco de dados
     */

    EXEC SQL BEGIN DECLARE SECTION;
```

```

int id;
VARCHAR nome[200];
EXEC SQL END DECLARE SECTION;

/*
 * Declaração a ser executada
 */

EXEC SQL DECLARE cursor_pessoal CURSOR FOR
    SELECT id, nome FROM pessoal ORDER BY nome;

/*
 * Executar a declaração
 */

EXEC SQL OPEN cursor_pessoal;

EXEC SQL WHENEVER NOT FOUND GOTO fechar_cursor;
while(1) /* a saída do laço é tratada pela declaração precedente */
{
    /* Ler o próximo valor */
    EXEC SQL FETCH cursor_pessoal INTO :id, :nome;
    printf("Lido: ID = %d; Nome = %s\n", id, nome.arr);
}

/* Limpeza */
fechar_cursor:
EXEC SQL CLOSE cursor_pessoal;
EXEC SQL DISCONNECT;

return 0;
}

```

Abaixo está mostrado o script utilizado para a pré-compilação, compilação, criação do executável e execução do programa. Como a árvore de fontes não foi instalada (`make install`), foi apenas construída (`make`), é necessário informar os diretórios dos cabeçalhos, bibliotecas e programas.

```

#!/bin/sh
# Script para pré-compilar, compilar e executar o programa 'pessoal.pgc'
ecpg pessoal.pgc
gcc pessoal.c -o pessoal \
-I ~/postgresql-8.0.0/src/interfaces/ecpg/include/ \
-I ~/postgresql-8.0.0/src/interfaces/libpq/ \
-I ~/postgresql-8.0.0/src/include/ \
-L ~/postgresql-8.0.0/src/interfaces/ecpg/ecpglib/ \
-l ecpg
./pessoal

```

Por fim está mostrado o resultado da execução do programa.

```

Lido: ID = 3; Nome = Francisco
Lido: ID = 2; Nome = Manuel
Lido: ID = 1; Nome = Maria

```

### Exemplo 29-2. Linguagem SQL estática incorporada

Este exemplo mostra a utilização de comandos SQL incorporados ao fonte do programa C. O programa estabelece uma conexão com o servidor de banco de dados por meio do TCP/IP, acessa o banco de dados “teste” através do usuário cujo nome é “teste” e cuja senha também é “teste”. Depois cria a tabela temporária `filmes`, carrega a tabela com os dados contidos na matriz `filmes`, e lista as linhas da tabela.

Abaixo está mostrado o código fonte do programa `sta-select.pgc`.

```

/*
 * Programa para mostrar a utilização do SQL estático incorporado.
 * Baseado em:
 *     Universidade de Toronto - DB2 - Embedded SQL Examples
 *     http://www.cs.toronto.edu/db/courses/db2/
 */

#include <stdio.h>

/*
 * Declarar as variáveis do programa
 */

int i;

/*
 * Declarar as variáveis hospedeiras
 */

EXEC SQL BEGIN DECLARE SECTION;
    /* Propriedades do filme */
    int    filme_id;
    char    filme_titulo[20];
    char    filme_diretor[20];
    /* Propriedades da conexão com o servidor de banco de dados */
    char *conexao="tcp:postgresql://127.0.0.1:5432/teste";
    char *usuario="teste";
    char *senha="teste";
    /* Filmes */
    struct filme {
        int    filme_id;
        char    filme_titulo[20];
        char    filme_diretor[20];
    };
    struct filme filmes[] = {
        {100,"Água Negra","Walter Salles"},
        {200,"Aprendendo a Mentir","Hendrik Handloegten"},
        {300,"Guerra dos Mundos","Steven Spielberg"},
        {400,"Quarteto Fantástico","Tim Story"}
    };
EXEC SQL END DECLARE SECTION;

/*
 * As linhas abaixo são redundantes, porque a ação padrão é continuar.
 * Estas linhas apenas mostram as situações que podem ocorrer,
 * e uma forma de controlá-las.
 */

                                /* sqlca.sqlcode == 0 (sem erro) */
EXEC SQL WHENEVER SQLWARNING CONTINUE; /* sqlca.sqlwarn[0] == 'W' */
EXEC SQL WHENEVER NOT FOUND CONTINUE; /* sqlca.sqlcode == ECPG_NOT_FOUND */

/*
 * Quando houver um erro executar a rotina de tratamento de erro.
 */

EXEC SQL WHENEVER SQLERROR DO erro(); /* sqlca.sqlcode < 0 */

/*
 * Rotina para tratamento de erro
 */

```

```

static void erro()
{ printf("#%ld:%s\n",sqlca.sqlcode,sqlca.sqlerrm.sqlerrmc);
  exit(1);
}

/*
 * Programa principal
 */

int main() {

/*
 * Conectar com o banco de dados
 */

EXEC SQL CONNECT TO :conexao USER :usuario IDENTIFIED BY :senha;

if (sqlca.sqlcode != 0) {
    printf("Falha na conexão!");
    erro();
}

/*
 * Criar a tabela filmes
 */

EXEC SQL CREATE TEMPORARY TABLE filmes (
    filme_id      INT NOT NULL PRIMARY KEY,
    filme_titulo  VARCHAR(20),
    filme_diretor  VARCHAR(20)
);

/*
 * Inserir as linhas na tabela
 */

for (i=0; i<(sizeof(filmes)/sizeof(filmes[0])); i++) {
    EXEC SQL INSERT INTO filmes VALUES(
        :filmes[i].filme_id,:filmes[i].filme_titulo,:filmes[i].filme_diretor
    );
}

/*
 * Imprimir o cabeçalho da tabela
 */

printf("+-----+-----+\n");
printf("| Nome do filme      | Diretor                |\n");
printf("+-----+-----+\n");

/*
 * Declarar um cursor para acessar as linhas da tabela
 */

EXEC SQL DECLARE c1 CURSOR FOR
    SELECT filme_titulo, filme_diretor
    FROM filmes;

/*
 * Abrir o cursor para ler as linhas da tabela
 */

```



```

EXEC SQL OPEN c1;

do {

    /*
     * Ler as linhas da tabela.
     * Executa a declaração implementada pelo cursor e retorna os resultados.
     */

    EXEC SQL FETCH c1 INTO :filme_titulo, :filme_diretor;
    if (SQLCODE != 0) break;          /* SQLCODE se refere a sqlca.sqlcode */

    /*
     * As variáveis hospedeiras devem ser prefixadas por ':'
     * quando são utilizadas nos comandos SQL
     */

    printf("| %-20s | %-20s |\n",filme_titulo,filme_diretor);

} while (1);

/*
 * Fechar a tabela
 */
printf("+-----+-----+\n");

/*
 * Fechar o cursor e a conexão
 */

EXEC SQL CLOSE c1;
EXEC SQL DISCONNECT;
}

```

Abaixo está mostrado o script utilizado para a pré-compilação, compilação, criação do executável e execução do programa. Como a árvore de fontes não foi instalada (make install), foi apenas construída (make), é necessário informar os diretórios dos cabeçalhos, bibliotecas e programas.

```

#!/bin/sh
# Script para pré-compilar, compilar e executar o programa 'sta-select.pgc'
~/postgresql-8.0.0/src/interfaces/ecpg/preproc/ecpg sta-select.pgc
gcc sta-select.c -o sta-select \
-I ~/postgresql-8.0.0/src/interfaces/ecpg/include/ \
-I ~/postgresql-8.0.0/src/interfaces/libpq/ \
-I ~/postgresql-8.0.0/src/include/ \
-L ~/postgresql-8.0.0/src/interfaces/ecpg/ecpglib/ \
-lecpq
./sta-select

```

Por fim está mostrado o resultado da execução do programa.

```

+-----+-----+
| Nome do filme      | Diretor                |
+-----+-----+
| Água Negra         | Walter Salles          |
| Aprendendo a Mentir | Hendrik Handloegten    |
| Guerra dos Mundos   | Steven Spielberg      |
| Quarteto Fantástico | Tim Story              |
+-----+-----+

```

### Exemplo 29-3. Linguagem SQL dinâmica incorporada

Este exemplo mostra a utilização de comandos SQL incorporados ao fonte do programa C. O programa estabelece uma conexão com o servidor de banco de dados por meio de soquete do domínio Unix, acessa o banco de dados “teste” através

do usuário cujo nome é “teste” e cuja senha também é “teste”. Depois cria a tabela temporária `filmes`, carrega a tabela com os dados contidos na matriz `filmes`, e lista as linhas da tabela. Se o nome do filme ou o nome do diretor for uma cadeia de caracteres vazia, os indicadores sinalizam um valor nulo.

Abaixo está mostrado o código fonte do programa `dyn-select.pgc`.

```
/*
 * Programa para mostrar a utilização do SQL dinâmico incorporado.
 * Baseado em:
 *     Universidade de Toronto - DB2 - Embedded SQL Examples
 *     http://www.cs.toronto.edu/db/courses/db2/
 */

#include <stdio.h>

/*
 * Declarar as variáveis do programa
 */

int i;

/*
 * Declarar as variáveis hospedeiras
 */

EXEC SQL BEGIN DECLARE SECTION;
    /* Propriedades do filme */
    int    filme_id;
    char   filme_titulo[20];
    char   filme_diretor[20];
    /* Propriedades da conexão com o servidor de banco de dados */
    char *conexao="unix:postgresql://localhost/teste";
    char *usuario="teste";
    char *senha="teste";
    /* Filmes */
    struct filme {
        int    filme_id;
        char   filme_titulo[20];
        char   filme_diretor[20];
    };
    struct filme filmes[] = {
        {100,"Água Negra","Walter Salles"},
        {200,"Aprendendo a Mentir","Hendrik Handloegten"},
        {300,"Guerra dos Mundos","Steven Spielberg"},
        {400,"Quarteto Fantástico","Tim Story"},
        {500,"Horror em Amityville",""}
    };
    /* Indicadores */
    int ind1, ind2, ind3;
    /* Declaração de inserção de filme */
    char *insere_filme = "INSERT INTO filmes VALUES(?, ?, ?)";
EXEC SQL END DECLARE SECTION;

/*
 * As linhas abaixo são redundantes, porque a ação padrão é continuar.
 * Estas linhas apenas mostram as situações que podem ocorrer,
 * e uma forma de controlá-las.
 */

/* sqlca.sqlcode == 0 (sem erro) */
EXEC SQL WHENEVER SQLWARNING CONTINUE; /* sqlca.sqlwarn[0] == 'W' */
EXEC SQL WHENEVER NOT FOUND CONTINUE; /* sqlca.sqlcode == ECPG_NOT_FOUND */
```

```

/*
 * Quando houver um erro executar a rotina de tratamento de erro.
 */

EXEC SQL WHENEVER SQLERROR DO erro(); /* sqlca.sqlcode < 0 */

/*
 * Rotina para tratamento de erro
 */

static void erro()
{ printf("#%ld:%s\n", sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
  exit(1);
}

/*
 * Programa principal
 */

int main() {

/*
 * Conectar com o banco de dados
 */

EXEC SQL CONNECT TO :conexao USER :usuario IDENTIFIED BY :senha;

if (sqlca.sqlcode != 0) {
    printf("Falha na conexão!");
    erro();
}

/*
 * Criar a tabela filmes
 */

EXEC SQL CREATE TEMPORARY TABLE filmes (
    filme_id      INT NOT NULL PRIMARY KEY,
    filme_titulo  VARCHAR(20),
    filme_diretor  VARCHAR(20)
);

/*
 * Inserir as linhas na tabela
 */

EXEC SQL PREPARE insere_filme FROM :insere_filme;

for (i=0; i<(sizeof(filmes)/sizeof(filmes[0])); i++) {
    ind1 = ind2 = ind3 = 0;
    if (strlen(filmes[i].filme_titulo) == 0) ind2=-1;
    if (strlen(filmes[i].filme_diretor) == 0) ind3=-1;
    EXEC SQL EXECUTE insere_filme USING
        :filmes[i].filme_id:ind1,
        :filmes[i].filme_titulo:ind2,
        :filmes[i].filme_diretor:ind3;
}

/*
 * Imprimir o cabeçalho da tabela
 */

```

```

printf("+-----+-----+\n");
printf("| Nome do filme      | Diretor                |\n");
printf("+-----+-----+\n");

/*
 * Declarar um cursor para acessar as linhas da tabela
 */

EXEC SQL DECLARE c1 CURSOR FOR
    SELECT filme_titulo, filme_diretor
    FROM filmes;

/*
 * Abrir o cursor para ler as linhas da tabela
 */

EXEC SQL OPEN c1;

do {

    /*
     * Ler as linhas da tabela.
     * Executa a declaração implementada pelo cursor e retorna os resultados.
     */

    EXEC SQL FETCH c1 INTO :filme_titulo:ind1, :filme_diretor:ind2;
    if (SQLCODE != 0) break; /* SQLCODE se refere a sqlca.sqlcode */
    /* Verificar valores nulos */
    if (ind1 == -1) {
        strcpy(filme_titulo, "-");
    }
    if (ind2 == -1) {
        strcpy(filme_diretor, "-");
    }

    /*
     * As variáveis hospedeiras devem ser prefixadas por ':'
     * quando são utilizadas nos comandos SQL
     */

    printf("| %-20s | %-20s |\n", filme_titulo, filme_diretor);

} while (1);

/*
 * Fechar a tabela
 */
printf("+-----+-----+\n");

/*
 * Fechar o cursor e a conexão
 */

EXEC SQL CLOSE c1;
EXEC SQL DISCONNECT;
}

```

Abaixo está mostrado o script utilizado para a pré-compilação, compilação, criação do executável e execução do programa. Como a árvore de fontes não foi instalada (`make install`), foi apenas construída (`make`), é necessário informar os diretórios dos cabeçalhos, bibliotecas e programas.

```
#!/bin/sh
```

```
# Script para pré-compilar, compilar e executar o programa 'dyn-select.pgc'
~/postgresql-8.0.0/src/interfaces/ecpg/preproc/ecpg dyn-select.pgc
gcc dyn-select.c -o dyn-select \
-I ~/postgresql-8.0.0/src/interfaces/ecpg/include/ \
-I ~/postgresql-8.0.0/src/interfaces/libpq/ \
-I ~/postgresql-8.0.0/src/include/ \
-L ~/postgresql-8.0.0/src/interfaces/ecpg/ecpglib/ \
-lecp
./dyn-select
```

Abaixo estão mostradas as linhas inseridas no começo do arquivo fonte C pelo pré-compilador.

```
/* Processed by ecp (3.2.0) */
/* These include files are added by the preprocessor */
#include <ecpgtype.h>
#include <ecpglib.h>
#include <ecpgerrno.h>
#include <sqlca.h>
/* End of automatic include section */
#line 1 "dyn-select.pgc"
```

Por fim está mostrado o resultado da execução do programa.

```
+-----+
| Nome do filme      | Diretor          |
+-----+
| Água Negra        | Walter Salles    |
| Aprendendo a Mentir | Hendrik Handloegten |
| Guerra dos Mundos  | Steven Spielberg |
| Quarteto Fantástico | Tim Story        |
| Horror em Amityville | -                |
+-----+
```

## Notas

1. DB2 — declaração válida da variável hospedeira `vstring`: `struct VARCHAR { short len; char s[10] } vstring;` Declaring host variables in C and C++ applications that use SQL (<http://publib.boulder.ibm.com/iserics/v5r2/ic2924/index.htm?info/rzajp/rzajpmst02.htm>) (N. do T.)
2. DB2 — se o valor da coluna de resultado for nulo, o SQL coloca -1 na variável indicadora; se não for utilizada uma variável indicadora, e a coluna de resultado tiver um valor nulo, retorna um SQLCODE negativo. Se o valor da coluna de resultado causar um erro de mapeamento, o SQL define a variável indicadora como -2. A variável indicadora também pode ser utilizada para verificar se o valor da cadeia de caracteres foi truncado. Quando há truncamento, a variável indicadora contém um inteiro positivo que especifica o comprimento original da cadeia de caracteres. (N. do T.)
3. DB2 — A variável indicadora vem imediatamente após a variável hospedeira. (N. do T.)

# Capítulo 30. O esquema de informações

O esquema de informações consiste em um conjunto de visões contendo informações sobre os objetos definidos no banco de dados corrente. O esquema de informações é definido no padrão SQL e, portanto, pode-se esperar que seja portátil e permaneça estável — isto é diferente dos catálogos do sistema, que são específicos do PostgreSQL e modelados segundo interesses da implementação. Entretanto, as visões do esquema de informações não contêm informações sobre funcionalidades específicas do PostgreSQL; para obter este tipo de informação, é necessário consultar os catálogos do sistema e outras visões específicas do PostgreSQL.<sup>1 2 3</sup>

## 30.1. O esquema

Em si, o esquema de informações é um esquema chamado `information_schema`. Este esquema é criado, automaticamente, em todos os bancos de dados. O dono deste esquema é o usuário de banco de dados inicial do agrupamento e, naturalmente, este usuário possui todos os privilégios neste esquema, incluindo a capacidade de removê-lo (mas o ganho de espaço obtido com a remoção é mínimo).

Por padrão, o esquema de informações não está no caminho de procura de esquemas e, portanto, todos os seus objetos devem ser acessados através de nomes qualificados. Uma vez que alguns objetos do esquema de informações possuem nomes genéricos, que podem existir também nos aplicativos dos usuários, deve-se tomar cuidado se for desejado colocar o esquema de informações no caminho de procura.

## 30.2. Tipos de dado

As colunas das visões do esquema de informações utilizam tipos de dado especiais, definidos no esquema de informações. São definidos como domínios simples sobre tipos nativos comuns. Estes tipos não devem ser utilizados fora do esquema de informações, mas os aplicativos devem estar preparados para trabalhar com os mesmos, caso façam consultas ao esquema de informações.

Os tipos são:

`cardinal_number`

Inteiro não negativo.<sup>4</sup>

`character_data`

Cadeia de caracteres (sem comprimento máximo especificado).<sup>5</sup>

`sql_identifier`

Cadeia de caracteres. Este tipo é utilizado para os identificadores do SQL, enquanto o tipo `character_data` é utilizado para todos os outros tipos de dado textuais.<sup>6</sup>

`time_stamp`

Domínio sobre o tipo `timestamp`<sup>7</sup>

Todas as colunas do esquema de informações possuem um destes quatro tipos.

O tipo de dado booleano (verdade/falso) é representado no esquema de informações por uma coluna do tipo `character_data` contendo YES ou NO (O esquema de informações foi inventado antes do tipo `boolean` ser adicionado ao padrão SQL e, portanto, esta convenção é necessária para manter o esquema de informações compatível com o passado).

**Exemplo:** Criação dos domínios do esquema de informações conforme o arquivo `src/backend/catalog/information_schema.sql`.<sup>8</sup>

```
=> CREATE DOMAIN cardinal_number AS INTEGER
->     CONSTRAINT cardinal_number_domain_check
->     CHECK (value >= 0 );
CREATE DOMAIN
=> CREATE DOMAIN character_data AS
->     CHARACTER VARYING;
CREATE DOMAIN
```

```
=> CREATE DOMAIN sql_identifier AS
->     CHARACTER VARYING;
CREATE DOMAIN
=> CREATE DOMAIN time_stamp AS TIMESTAMP(2)
->     DEFAULT current_timestamp(2);
CREATE DOMAIN
```

### 30.3. information\_schema\_catalog\_name

information\_schema\_catalog\_name é uma tabela que sempre possui uma linha e uma coluna contendo o nome do banco de dados corrente (catálogo corrente, na terminologia SQL).<sup>9</sup>

**Tabela 30-1. Colunas de information\_schema\_catalog\_name**

Nome	Tipo de dado	Descrição
catalog_name	sql_identifier	Nome do banco de dados que contém este esquema de informações

**Exemplo:** Consultar o nome do catálogo estando conectado ao banco de dados teste:<sup>10</sup>

```
teste=> SELECT * FROM information_schema.information_schema_catalog_name;
```

```
catalog_name
-----
teste
(1 linha)
```

```
teste=> SELECT current_database();
```

```
current_database
-----
teste
(1 linha)
```

### 30.4. applicable\_roles

A visão applicable\_roles identifica todos os grupos que o usuário corrente é membro (Uma role é a mesma coisa que um grupo). Geralmente é melhor utilizar a visão enabled\_roles em vez desta visão.<sup>11</sup>

**Tabela 30-2. Colunas de applicable\_roles**

Nome	Tipo de dado	Descrição
grantee	sql_identifier	Sempre o nome do usuário corrente
role_name	sql_identifier	Nome do grupo
is_grantable	character_data	Se aplica a uma funcionalidade não disponível no PostgreSQL

**Exemplo:** Criar os grupos engenharia e arquitetura, tornando o usuário teste membro destes grupos.<sup>12</sup>

```
teste=> \c teste postgres
Conectado ao banco de dados "teste" como usuário "postgres".
teste=# CREATE GROUP engenharia WITH USER teste;
teste=# CREATE GROUP arquitetura WITH USER teste;
teste=# \c teste teste
Conectado ao banco de dados "teste" como usuário "teste".
teste=> SELECT * FROM information_schema.applicable_roles;
```

```

grantee | role_name | is_grantable
-----+-----+-----
teste  | engenharia | NO
teste  | arquitetura | NO
(2 linhas)

```

```
teste=> \dg
```

```

          Lista de grupos
Nome do Grupo | ID do Grupo
-----+-----
arquitetura   |          101
engenharia    |          100

```

```
teste=> SELECT * FROM information_schema.enabled_roles;
```

```

role_name
-----
engenharia
arquitetura
(2 linhas)

```

### 30.5. check\_constraints

A visão `check_constraints` contém todas as restrições de verificação, tanto as definidas nas tabelas quanto as definidas nos domínios, que pertencem ao usuário corrente (O dono da tabela ou do domínio é o dono da restrição).<sup>13 14 15</sup>

**Tabela 30-3. Colunas de `check_constraints`**

Nome	Tipo de dado	Descrição
<code>constraint_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a restrição (sempre o banco de dados corrente)
<code>constraint_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a restrição
<code>constraint_name</code>	<code>sql_identifier</code>	Nome da restrição
<code>check_clause</code>	<code>character_data</code>	A expressão de verificação da restrição de verificação

**Exemplo:** Consultar a visão `check_constraints`.<sup>16</sup>

```
=> SELECT * FROM information_schema.check_constraints;
```

```

constraint_catalog | constraint_schema | constraint_name | check_clause
-----+-----+-----+-----
teste              | public            | cardinal_number_domain_check | ((VALUE >= 0))
(1 linha)

```

### 30.6. column\_domain\_usage

A visão `column_domain_usage` identifica todas as colunas (da tabela ou da visão) que fazem uso de algum domínio definido no banco de dados corrente e que pertence ao usuário corrente.<sup>17 18 19</sup>



**Tabela 30-4. Colunas de `column_domain_usage`**

Nome	Tipo de dado	Descrição
<code>domain_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém o domínio (sempre o banco de dados corrente)
<code>domain_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém o domínio
<code>domain_name</code>	<code>sql_identifier</code>	Nome do domínio
<code>table_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a tabela (sempre o banco de dados corrente)
<code>table_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a tabela
<code>table_name</code>	<code>sql_identifier</code>	Nome da tabela
<code>column_name</code>	<code>sql_identifier</code>	Nome da coluna

**Exemplo:** Consultar a visão `column_domain_usage`.<sup>20</sup>

```
=> CREATE TABLE tbl_cardinal(cardinal CARDINAL_NUMBER PRIMARY KEY);
```

NOTA: CREATE TABLE / PRIMARY KEY criará índice implícito "tbl\_cardinal\_pkey" na tabela "tbl\_cardinal"

```
=> SELECT * FROM information_schema.column_domain_usage;
```

```
domain_catalog | domain_schema | domain_name | table_catalog | table_schema | table_name |
column_name
-----+-----+-----+-----+-----+-----+-----
teste          | public        | cardinal_number | teste          | public        | tbl_cardinal |
cardinal
(1 linha)
```

## 30.7. column\_privileges

A visão `column_privileges` identifica todos os privilégios concedidos em colunas para o usuário corrente, ou pelo usuário corrente. Existe uma linha para cada combinação de coluna, quem concedeu e a quem foi concedido. Os privilégios concedidos aos grupos são identificados na visão `role_column_grants`.

No PostgreSQL só é possível conceder privilégios para toda a tabela, e não para colunas individuais. Portanto, esta visão contém as mesmas informações da visão `table_privileges`, apenas representadas por uma linha para cada coluna da respectiva tabela, mas somente abrangendo os tipos de privilégio onde a granularidade de coluna é possível: `SELECT`, `INSERT`, `UPDATE` e `REFERENCES`. Se for desejado tornar os aplicativos adequados para possíveis desenvolvimentos futuros, geralmente é correto escolher esta visão em vez de `table_privileges`, quando há interesse em um destes tipos de privilégio.<sup>21 22 23</sup>

**Tabela 30-5. Colunas de `column_privileges`**

Nome	Tipo de dado	Descrição
<code>grantor</code>	<code>sql_identifier</code>	Nome do usuário que concedeu o privilégio
<code>grantee</code>	<code>sql_identifier</code>	Nome do usuário ou do grupo para o qual o privilégio foi concedido
<code>table_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados contendo a tabela que contém a coluna (sempre o banco de dados corrente)
<code>table_schema</code>	<code>sql_identifier</code>	Nome do esquema contendo a tabela que contém a coluna
<code>table_name</code>	<code>sql_identifier</code>	Nome da tabela que contém a coluna

Nome	Tipo de dado	Descrição
column_name	sql_identifier	Nome da coluna
privilege_type	character_data	Tipo do privilégio: SELECT, INSERT, UPDATE ou REFERENCES
is_grantable	character_data	YES se o privilégio pode ser concedido, NO caso contrário

Deve ser observado que a coluna grantee não faz distinção entre usuários e grupos. Havendo usuários e grupos com o mesmo nome, infelizmente não há maneira de distingui-los. Possivelmente será proibido existir usuários e grupos com o mesmo nome em uma versão futura do PostgreSQL.

## 30.8. column\_udt\_usage

A visão `column_udt_usage` identifica todas as colunas que utilizam tipos de dado pertencentes ao usuário corrente. Deve ser observado que no PostgreSQL os tipos de dado nativos se comportam como os tipos definidos pelo usuário e, portanto, também são incluídos. Para obter mais detalhes consulte também a Seção 30.9.<sup>24</sup>

**Tabela 30-6. Colunas de `column_udt_usage`**

Nome	Tipo de dado	Descrição
udt_catalog	sql_identifier	Nome do banco de dados onde o tipo de dado da coluna (o tipo subjacente do domínio, quando aplicável) está definido (sempre o banco de dados corrente)
udt_schema	sql_identifier	Nome do esquema onde o tipo de dado da coluna (o tipo subjacente do domínio, quando aplicável) está definido
udt_name	sql_identifier	Nome do tipo de dado da coluna (o tipo subjacente do domínio, quando aplicável)
table_catalog	sql_identifier	Nome do banco de dados que contém a tabela (sempre o banco de dados corrente)
table_schema	sql_identifier	Nome do esquema que contém a tabela
table_name	sql_identifier	Nome da tabela
column_name	sql_identifier	Nome da coluna

## 30.9. columns

A visão `columns` mostra informações sobre todas as colunas das tabelas (ou colunas das visões) do banco de dados. As colunas do sistema (`oid`, etc.) não são incluídas. Somente são mostradas as colunas que o usuário corrente tem acesso (por ser o dono ou por ter algum privilégio).<sup>25 26 27</sup>

**Tabela 30-7. Colunas de `columns`**

Nome	Tipo de dado	Descrição
table_catalog	sql_identifier	Nome do banco de dados que contém a tabela (sempre o banco de dados corrente)
table_schema	sql_identifier	Nome do esquema que contém a tabela
table_name	sql_identifier	Nome da tabela
column_name	sql_identifier	Nome da coluna
ordinal_position	cardinal_number	Posição ordinal da coluna na tabela (contada a partir de 1)
column_default	character_data	Expressão padrão da coluna (nulo se o usuário corrente não for o dono da tabela que contém a coluna)

Nome	Tipo de dado	Descrição
is_nullable	character_data	YES se a coluna puder ser nula, NO se não puder ser nula. A restrição de não-nulo é uma maneira de saber que a coluna não pode ser nula, mas podem haver outras.
data_type	character_data	Tipo de dado da coluna, se for um tipo nativo, ARRAY se for uma matriz (neste caso deve ser vista a visão <code>element_types</code> ), ou então USER-DEFINED (neste caso o tipo é identificado em <code>udt_name</code> e nas colunas associadas). Se a coluna for baseada em um domínio, esta coluna se refere ao tipo subjacente do domínio (e o domínio é identificado em <code>domain_name</code> e nas colunas associadas).
character_maximum_length	cardinal_number	Quando <code>data_type</code> identifica um tipo cadeia de caracteres ou de bits, o comprimento máximo declarado; nulo para todos os outros tipos de dado, ou se o comprimento máximo não for declarado.
character_octet_length	cardinal_number	Quando <code>data_type</code> identifica um tipo caractere, o comprimento máximo possível em octetos (bytes) do dado (não deve ser de interesse dos usuários do PostgreSQL); nulo para todos os outros tipos de dado.
numeric_precision	cardinal_number	Quando <code>data_type</code> identifica um tipo numérico, esta coluna contém a precisão (declarada ou implícita) do tipo de dado desta coluna. A precisão indica o número de dígitos significativos. Pode ser expresso em termos decimais (base 10), ou binários (base 2), conforme especificado na coluna <code>numeric_precision_radix</code> . Para todos ou outros tipos de dado, esta coluna é nula.
numeric_precision_radix	cardinal_number	Quando <code>data_type</code> identifica um tipo numérico, esta coluna indica em que base estão expressos os valores nas colunas <code>numeric_precision</code> e <code>numeric_scale</code> . O valor é 2 ou 10. Para todos ou outros tipos de dado, esta coluna é nula.
numeric_scale	cardinal_number	Quando <code>data_type</code> identifica um tipo numérico exato, esta coluna contém a escala (declarada ou implícita) do tipo desta coluna. A escala indica o número de dígitos significativos à direita do ponto decimal. Pode ser expresso em termos decimais (base 10), ou binários (base 2), conforme especificado na coluna <code>numeric_precision_radix</code> . Para todos ou outros tipos de dado, esta coluna é nula.
datetime_precision	cardinal_number	Quando <code>data_type</code> identifica um tipo data, hora ou intervalo, a precisão declarada; nulo para todos os outros tipos de dado, ou se a precisão não for declarada.
interval_type	character_data	Ainda não implementado
interval_precision	character_data	Ainda não implementado
character_set_catalog	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
character_set_schema	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
character_set_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
collation_catalog	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
collation_schema	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL

Nome	Tipo de dado	Descrição
collation_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
domain_catalog	sql_identifier	Quando a coluna possui um tipo domínio, o nome do banco de dados onde o domínio está definido (sempre o banco de dados corrente), senão nulo.
domain_schema	sql_identifier	Quando a coluna possui um tipo domínio, o nome do esquema onde o domínio está definido, senão nulo.
domain_name	sql_identifier	Quando a coluna possui um tipo domínio, o nome do domínio, senão nulo.
udt_catalog	sql_identifier	Nome do banco de dados onde o tipo de dado da coluna (o tipo subjacente do domínio, quando aplicável) está definido (sempre o banco de dados corrente) (udt = tipo definido pelo usuário. N. do T.)
udt_schema	sql_identifier	Nome do esquema onde o tipo de dado da coluna (o tipo subjacente do domínio, quando aplicável) está definido
udt_name	sql_identifier	Nome tipo de dado da coluna (o tipo subjacente do domínio, quando aplicável)
scope_catalog	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
scope_schema	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
scope_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
maximum_cardinality	cardinal_number	Sempre nulo, porque as matrizes sempre possuem uma cardinalidade <sup>a</sup> máxima não definida no PostgreSQL
dtd_identifier	sql_identifier	O identificador do descritor do tipo de dado da coluna, único entre todos os descritores <sup>b</sup> de tipo de dado pertencentes à tabela. A utilidade principal é para fazer junção com outras instâncias destes identificadores (O formato específico do identificador não é definido, e também não há garantia que permaneça o mesmo nas versões futuras).
is_self_referencing	character_data	Se aplica a uma funcionalidade não disponível no PostgreSQL
<p>Notas:</p> <p>a. cardinalidade — O número de elementos da coleção. Os elementos não precisam necessariamente possuir valores distintos. Os objetos aos quais este conceito se aplica incluem as tabelas e os valores dos tipos coleção. (ISO-ANSI Working Draft) Foundation (SQL/Foundation), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E).(N. do T.)</p> <p>b. descritor — Uma descrição codificada de um objeto SQL. Inclui todas as informações sobre o objeto requeridas por uma implementação em conformidade com o SQL. (Second Informal Review Draft) ISO/IEC 9075:1992, Database Language SQL- July 30, 1992 (N. do T.)</p>		

Uma vez que os tipos de dado podem ser definidos de várias maneiras no SQL, e o PostgreSQL possui maneiras adicionais de definir tipos de dado, sua representação no esquema de informações pode ser um tanto difícil. Supõe-se que o `data_type` da coluna identifique o tipo nativo subjacente da coluna. No PostgreSQL isto significa que o tipo é definido no catálogo do sistema `pg_catalog`. Esta coluna pode ser útil se o aplicativo puder tratar de forma especial os tipos de dado nativos bem conhecidos (por exemplo, formatar tipos numéricos de forma diferente, ou utilizar os dados nas colunas de precisão). As colunas `udt_name`, `udt_schema` e `udt_catalog` sempre identificam o tipo de dado subjacente da coluna, mesmo quando a coluna é baseada em um domínio (Uma vez que o PostgreSQL trata os tipos nativos da mesma maneira que os tipos definidos pelo usuário, os tipos nativos também aparecem aqui. É uma extensão ao padrão SQL). Estas colunas devem ser utilizadas se o aplicativo desejar processar os dados de forma diferente conforme o tipo, porque neste caso não vai importar se a coluna realmente se baseia em um domínio. Se a coluna for baseada em um domínio, a identidade do

domínio é armazenada nas colunas `domain_name`, `domain_schema` e `domain_catalog`. Se for desejado agrupar as colunas com seus tipos de dado associados e tratar os domínios como tipos separados, pode ser utilizado `coalesce(domain_name, udt_name)`, etc.

**Exemplo:** Consultar a visão `columns`.<sup>28</sup>

```
=> \x
```

Habilitada a exibição expandida.

```
=> SELECT * FROM information_schema.columns WHERE table_schema='public';
```

```
-[ LINHA 1 ]-----+-----
table_catalog      | teste
table_schema      | public
table_name        | tbl_cardinal
column_name       | cardinal
ordinal_position  | 1
column_default    |
is_nullable      | NO
data_type         | integer
character_maximum_length |
character_octet_length |
numeric_precision | 32 <- Precisão de 32 bits
numeric_precision_radix | 2 <- Sinaliza binário (bits)
numeric_scale     | 0
datetime_precision |
interval_type     |
interval_precision |
character_set_catalog |
character_set_schema |
character_set_name  |
collation_catalog  |
collation_schema    |
collation_name      |
domain_catalog     | teste
domain_schema      | public
domain_name        | cardinal_number
udt_catalog        | teste
udt_schema         | pg_catalog
udt_name           | int4
scope_catalog      |
scope_schema       |
scope_name         |
maximum_cardinality |
dtd_identifier     | 1
is_self_referencing | NO
```

### 30.10. constraint\_column\_usage

A visão `constraint_column_usage` identifica todas as colunas no banco de dados corrente utilizadas por alguma restrição. Somente são mostradas as colunas contidas nas tabelas que pertencem ao usuário corrente. Para as restrições de verificação, esta visão identifica as colunas utilizadas na expressão de verificação. Para as restrições de chave estrangeira, esta visão identifica as colunas referenciadas pela chave estrangeira. Para as restrições de unicidade ou de chave primária, esta visão identifica as colunas restringidas.<sup>29 30 31</sup>

**Tabela 30-8. Colunas de `constraint_column_usage`**

Nome	Tipo de dado	Descrição
<code>table_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados contendo a tabela que contém a coluna utilizada por alguma restrição (sempre o banco de dados corrente)
<code>table_schema</code>	<code>sql_identifier</code>	Nome do esquema contendo a tabela que contém a coluna utilizada por alguma restrição
<code>table_name</code>	<code>sql_identifier</code>	Nome da tabela que contém a coluna utilizada por alguma restrição
<code>column_name</code>	<code>sql_identifier</code>	Nome da coluna utilizada por alguma restrição
<code>constraint_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a restrição (sempre o banco de dados corrente)
<code>constraint_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a restrição
<code>constraint_name</code>	<code>sql_identifier</code>	Nome da restrição

**Exemplo:** Consultar a visão `constraint_column_usage`.<sup>32</sup>

```
=> \x
```

Habilitada a exibição expandida.

```
=> SELECT * FROM information_schema.constraint_column_usage;
```

```
-[ LINHA 1 ]-----+-----
table_catalog      | teste
table_schema       | public
table_name         | tbl_cardinal
column_name        | cardinal
constraint_catalog | teste
constraint_schema  | public
constraint_name     | tbl_cardinal_pkey
```

### 30.11. `constraint_table_usage`

A visão `constraint_table_usage` identifica todas as tabelas do banco de dados corrente utilizadas por alguma restrição, e que pertencem ao usuário corrente (Esta visão é diferente da visão `table_constraints`, que identifica todas as restrições de tabela junto com a tabela onde são definidas). Para as restrições de chave estrangeira, esta visão identifica a tabela referenciada pela chave estrangeira. Para as restrições de unicidade e de chave primária, esta visão simplesmente identifica qual tabela a restrição pertence. As restrições de verificação e de não-nulo não são incluídas nesta visão.<sup>33 34 35</sup>

**Tabela 30-9. Colunas de `constraint_table_usage`**

Nome	Tipo de dado	Descrição
<code>table_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a tabela utilizada por alguma restrição (sempre o banco de dados corrente)
<code>table_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a tabela utilizada por alguma restrição
<code>table_name</code>	<code>sql_identifier</code>	Nome da tabela utilizada por alguma restrição
<code>constraint_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a restrição (sempre o banco de dados corrente)
<code>constraint_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a restrição
<code>constraint_name</code>	<code>sql_identifier</code>	Nome da restrição

**Exemplo:** Consultar a visão `constraint_table_usage`.<sup>36</sup>

```
=> \x
```

Habilitada a exibição expandida.

```
=> SELECT * FROM information_schema.constraint_table_usage;
```

```
-[ LINHA 1 ]-----+-----
table_catalog      | teste
table_schema       | public
table_name         | tbl_cardinal
constraint_catalog  | teste
constraint_schema  | public
constraint_name     | tbl_cardinal_pkey
```

## 30.12. data\_type\_privileges

A visão `data_type_privileges` identifica todos os descritores de tipo de dado que o usuário corrente pode acessar, seja por ser o dono do objeto descrito ou por ter algum privilégio sobre o mesmo. O descritor de tipo de dado é gerado sempre que o tipo de dado é utilizado na definição da coluna de uma tabela, de um domínio ou de uma função (como parâmetro ou tipo retornado), e armazena alguma informação sobre como o tipo de dado é utilizado nesta instância (por exemplo, o comprimento máximo declarado, se aplicável). Para cada descritor de tipo de dado é atribuído um identificador arbitrário que é único entre os identificadores de descritor de tipo de dado atribuídos para um objeto (tabela, domínio, função). Provavelmente esta visão não tem utilidade para os aplicativos, mas é utilizada para definir algumas outras visões do esquema de informações.

**Tabela 30-10. Colunas de `data_type_privileges`**

Nome	Tipo de dado	Descrição
<code>object_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém o objeto descrito (sempre o banco de dados corrente)
<code>object_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém o objeto descrito
<code>object_name</code>	<code>sql_identifier</code>	Nome do objeto descrito
<code>object_type</code>	<code>character_data</code>	Tipo do objeto descrito: um entre <code>TABLE</code> (o descritor do tipo de dado pertence a uma coluna desta tabela), <code>DOMAIN</code> (o descritor do tipo de dado pertence a este domínio) ou <code>ROUTINE</code> (o descritor do tipo de dado pertence ao tipo de dado de um parâmetro ou do valor retornado pela função).
<code>dtd_identifier</code>	<code>sql_identifier</code>	O identificador do descritor do tipo de dado, que é único entre os descritores de tipo de dado para o mesmo objeto.

## 30.13. domain\_constraints

A visão `domain_constraints` contém todas as restrições pertencentes aos domínios que pertencem ao usuário corrente.

37 38

**Tabela 30-11. Colunas de `domain_constraints`**

Nome	Tipo de dado	Descrição
<code>constraint_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a restrição (sempre o banco de dados corrente)
<code>constraint_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a restrição
<code>constraint_name</code>	<code>sql_identifier</code>	Nome da restrição
<code>domain_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém o domínio (sempre o banco de dados corrente)

Nome	Tipo de dado	Descrição
domain_schema	sql_identifier	Nome do esquema que contém o domínio
domain_name	sql_identifier	Nome do domínio
is_deferrable	character_data	YES se a restrição for postergável, NO caso contrário
initially_deferred	character_data	YES se a restrição for postergável e inicialmente postergada, NO caso contrário

**Exemplo:** Consultar a visão `domain_constraints`.<sup>39</sup>

```
=> \x
```

Habilitada a exibição expandida.

```
=> SELECT * FROM information_schema.domain_constraints;
```

```
-[ LINHA 1 ]-----+-----
constraint_catalog | teste
constraint_schema | public
constraint_name    | cardinal_number_domain_check
domain_catalog     | teste
domain_schema      | public
domain_name        | cardinal_number
is_deferrable      | NO
initially_deferred | NO
```

### 30.14. domain\_udt\_usage

A visão `domain_udt_usage` identifica todas as colunas que utilizam um tipo de dado pertencente ao usuário corrente. Deve ser observado que no PostgreSQL os tipos de dado nativos se comportam como os tipos definidos pelo usuário e, portanto, também são incluídos.

**Tabela 30-12.** Colunas de `domain_udt_usage`

Nome	Tipo de dado	Descrição
udt_catalog	sql_identifier	Nome do banco de dados onde o tipo de dado do domínio está definido (sempre o banco de dados corrente)
udt_schema	sql_identifier	Nome do esquema onde o tipo de dado do domínio está definido
udt_name	sql_identifier	Nome do tipo de dado do domínio
domain_catalog	sql_identifier	Nome do banco de dados que contém o domínio (sempre o banco de dados corrente)
domain_schema	sql_identifier	Nome do esquema que contém o domínio
domain_name	sql_identifier	Nome do domínio

### 30.15. domains

A visão `domains` contém todos os domínios definidos no banco de dados corrente.<sup>40 41</sup>

**Tabela 30-13.** Colunas de `domains`

Nome	Tipo de dado	Descrição
domain_catalog	sql_identifier	Nome do banco de dados que contém o domínio (sempre o banco de dados corrente)



Nome	Tipo de dado	Descrição
domain_schema	sql_identifier	Nome do esquema que contém o domínio
domain_name	sql_identifier	Nome do domínio
data_type	character_data	Tipo de dado do domínio, se for um tipo nativo, ou ARRAY se for uma matriz (neste caso deve ser consultada a visão <code>element_types</code> ), ou então USER-DEFINED (neste caso o tipo é identificado em <code>udt_name</code> e nas colunas associadas).
character_maximum_length	cardinal_number	Quando o domínio tem um tipo cadeia de caracteres ou de bits, o comprimento máximo declarado; nulo para todos os outros tipos de dado, ou se o comprimento máximo não for declarado.
character_octet_length	cardinal_number	Quando o domínio tem um tipo caractere, o comprimento máximo possível em octetos (bytes) do dado (não deve ser de interesse dos usuários do PostgreSQL); nulo para todos os outros tipos de dado.
character_set_catalog	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
character_set_schema	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
character_set_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
collation_catalog	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
collation_schema	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
collation_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
numeric_precision	cardinal_number	Quando o domínio possui um tipo numérico, esta coluna contém a precisão (declarada ou implícita) do tipo de dado desta coluna. A precisão indica o número de dígitos significativos. Pode ser expresso em termos decimais (base 10), ou binários (base 2), conforme especificado na coluna <code>numeric_precision_radix</code> . Para todos ou outros tipos de dado, esta coluna é nula.
numeric_precision_radix	cardinal_number	Quando o domínio tem um tipo numérico, esta coluna indica em que base estão expressos os valores nas colunas <code>numeric_precision</code> e <code>numeric_scale</code> . O valor é 2 ou 10. Para todos ou outros tipos de dado, esta coluna é nula.
numeric_scale	cardinal_number	Quando o domínio possui um tipo numérico exato, esta coluna contém a escala (declarada ou implícita) do tipo desta coluna. A escala indica o número de dígitos significativos à direita do ponto decimal. Pode ser expresso em termos decimais (base 10), ou binários (base 2), conforme especificado na coluna <code>numeric_precision_radix</code> . Para todos ou outros tipos de dado, esta coluna é nula.
datetime_precision	cardinal_number	Quando o domínio tem um tipo data, hora ou intervalo, a precisão declarada; nulo para todos os outros tipos de dado, ou se a precisão não for declarada.
interval_type	character_data	Ainda não implementado
interval_precision	character_data	Ainda não implementado
domain_default	character_data	Expressão padrão do domínio

Nome	Tipo de dado	Descrição
udt_catalog	sql_identifier	Nome do banco de dados onde o tipo de dado do domínio está definido (sempre o banco de dados corrente)
udt_schema	sql_identifier	Nome do esquema onde o tipo de dado do domínio está definido
udt_name	sql_identifier	Nome do tipo de dado do domínio
scope_catalog	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
scope_schema	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
scope_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
maximum_cardinality	cardinal_number	Sempre nulo, porque a matrizes sempre possuem uma cardinalidade máxima não definida no PostgreSQL
dtd_identifier	sql_identifier	O identificador do descritor do tipo de dado do domínio, único entre os descritores de tipo de dado pertencentes ao domínio (o que é óbvio, porque o domínio somente contém um descritor de tipo de dado). A utilidade principal é para fazer junção com outras instâncias destes identificadores (O formato específico do identificador não é definido, e também não há garantia que permaneça o mesmo nas versões futuras).

**Exemplo:** Consultar a visão domains.<sup>42</sup>

```
=> \x
```

Habilitada a exibição expandida.

```
=> SELECT * FROM information_schema.domains WHERE domain_schema='public';
```

```
-[ LINHA 1 ]-----+-----
domain_catalog      | teste
domain_schema       | public
domain_name         | cardinal_number
data_type           | integer
character_maximum_length |
character_octet_length |
character_set_catalog |
character_set_schema |
character_set_name   |
collation_catalog   |
collation_schema     |
collation_name       |
numeric_precision    | 32 <- Precisão de 32 bits
numeric_precision_radix | 2 <- Sinaliza binário (bits)
numeric_scale        | 0
datetime_precision  |
interval_type        |
interval_precision   |
domain_default       |
udt_catalog          | teste
udt_schema           | pg_catalog
udt_name             | int4
scope_catalog        |
scope_schema         |
scope_name           |
maximum_cardinality  |
dtd_identifier       | 1
```

```

-[ LINHA 2 ]-----+-----
domain_catalog      | teste
domain_schema       | public
domain_name         | character_data
data_type           | character varying
character_maximum_length |
character_octet_length | 1073741824
character_set_catalog |
character_set_schema |
character_set_name   |
collation_catalog    |
collation_schema     |
collation_name       |
numeric_precision    |
numeric_precision_radix |
numeric_scale        |
datetime_precision   |
interval_type        |
interval_precision   |
domain_default       |
udt_catalog          | teste
udt_schema           | pg_catalog
udt_name             | varchar
scope_catalog        |
scope_schema         |
scope_name           |
maximum_cardinality  |
dtd_identifier       | 1
-[ LINHA 3 ]-----+-----
domain_catalog      | teste
domain_schema       | public
domain_name         | sql_identifier
data_type           | character varying
character_maximum_length |
character_octet_length | 1073741824
character_set_catalog |
character_set_schema |
character_set_name   |
collation_catalog    |
collation_schema     |
collation_name       |
numeric_precision    |
numeric_precision_radix |
numeric_scale        |
datetime_precision   |
interval_type        |
interval_precision   |
domain_default       |
udt_catalog          | teste
udt_schema           | pg_catalog
udt_name             | varchar
scope_catalog        |
scope_schema         |
scope_name           |
maximum_cardinality  |
dtd_identifier       | 1
-[ LINHA 4 ]-----+-----
domain_catalog      | teste
domain_schema       | public
domain_name         | time_stamp
data_type           | timestamp without time zone
character_maximum_length |

```

```

character_octet_length |
character_set_catalog  |
character_set_schema   |
character_set_name     |
collation_catalog      |
collation_schema       |
collation_name         |
numeric_precision      |
numeric_precision_radix |
numeric_scale          |
datetime_precision    | 2
interval_type          |
interval_precision     |
domain_default         | ('now'::text)::timestamp(2) with time zone
udt_catalog            | teste
udt_schema              | pg_catalog
udt_name                | timestamp
scope_catalog          |
scope_schema           |
scope_name             |
maximum_cardinality    |
dtd_identifier         | 1

```

```
=> \x
```

Desabilitada a exibição expandida.

```
=> \dD
```

```

                                Lista de domínios
Esquema | Nome | Tipo | Modificador
-----+-----+-----+-----
public | cardinal_number | integer |
public | character_data | character varying |
public | sql_identifier | character varying |
public | time_stamp | timestamp(2) without time zone | default ('now'::text)::timestamp(2) with
time zone
(4 linhas)

```

## 30.16. element\_types

A visão `element_types` contém os descritores do tipo de dado dos elementos das matrizes. Quando uma coluna de tabela, domínio, parâmetro de função ou valor retornado por uma função é definido como sendo do tipo matriz, a visão do esquema de informações respectiva somente contém `ARRAY` na coluna `data_type`. Para obter informações sobre o tipo do elemento da matriz pode ser feita a junção da visão respectiva com esta visão. Por exemplo, para mostrar as colunas da tabela com os tipos de dado e tipos dos elementos da matriz, se aplicável, pode ser executado:

```

SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN information_schema.element_types e
    ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE', c.dtd_identifier)
        = (e.object_catalog, e.object_schema, e.object_name, e.object_type,
            e.array_type_identifier))
WHERE c.table_schema = '...' AND c.table_name = '...'
ORDER BY c.ordinal_position;

```

Esta visão somente inclui os objetos que o usuário corrente pode acessar, seja por ser o dono ou por possuir algum privilégio.<sup>43</sup>

Tabela 30-14. Colunas de `element_types`

Nome	Tipo de dado	Descrição
<code>object_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém o objeto que utiliza a matriz sendo descrita (sempre o banco de dados corrente)
<code>object_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém o objeto que utiliza a matriz sendo descrita
<code>object_name</code>	<code>sql_identifier</code>	Nome do objeto que utiliza a matriz sendo descrita
<code>object_type</code>	<code>character_data</code>	O tipo do objeto que utiliza a matriz sendo descrita: um entre <code>TABLE</code> (a matriz é utilizada por uma coluna desta tabela), <code>DOMAIN</code> (a matriz é utilizada por este domínio), <code>ROUTINE</code> (a matriz é utilizada pelo tipo de dado de um parâmetro ou do valor retornado, desta função).
<code>array_type_identifier</code>	<code>sql_identifier</code>	O identificador do descritor do tipo de dado da matriz sendo descrita. Utilizado para fazer junção com as colunas <code>dtd_identifier</code> de outras visões do esquema de informações.
<code>data_type</code>	<code>character_data</code>	Tipo de dado dos elementos da matriz, se for um tipo nativo, senão <code>USER-DEFINED</code> (neste caso, o tipo é identificado em <code>udt_name</code> e nas colunas associadas).
<code>character_maximum_length</code>	<code>cardinal_number</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de elementos de matriz no PostgreSQL
<code>character_octet_length</code>	<code>cardinal_number</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de elementos de matriz no PostgreSQL
<code>character_set_catalog</code>	<code>sql_identifier</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>character_set_schema</code>	<code>sql_identifier</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>character_set_name</code>	<code>sql_identifier</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>collation_catalog</code>	<code>sql_identifier</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>collation_schema</code>	<code>sql_identifier</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>collation_name</code>	<code>sql_identifier</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>numeric_precision</code>	<code>cardinal_number</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de elementos de matriz no PostgreSQL
<code>numeric_precision_radix</code>	<code>cardinal_number</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de elementos de matriz no PostgreSQL
<code>numeric_scale</code>	<code>cardinal_number</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de elementos de matriz no PostgreSQL
<code>datetime_precision</code>	<code>cardinal_number</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de elementos de matriz no PostgreSQL
<code>interval_type</code>	<code>character_data</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de elementos de matriz no PostgreSQL
<code>interval_precision</code>	<code>character_data</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de elementos de matriz no PostgreSQL
<code>domain_default</code>	<code>character_data</code>	Ainda não implementado
<code>udt_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados onde o tipo de dado dos elementos

Nome	Tipo de dado	Descrição
		está definido (sempre o banco de dados corrente)
udt_schema	sql_identifier	Nome do esquema onde o tipo de dado dos elementos está definido
udt_name	sql_identifier	Nome do tipo de dado dos elementos
scope_catalog	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
scope_schema	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
scope_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
maximum_cardinality	cardinal_number	Sempre nulo, porque as matrizes sempre possuem uma cardinalidade máxima não definida no PostgreSQL
dtd_identifier	sql_identifier	O identificador do descritor do tipo de dado do elemento. Atualmente não tem utilidade.

### 30.17. enabled\_roles

A visão `enabled_roles` identifica todos os grupos que o usuário corrente é membro (Uma `role` é a mesma coisa que um grupo). A diferença entre esta visão e a visão `applicable_roles` é que, no futuro, poderá haver um mecanismo para habilitar e desabilitar grupos durante a sessão. Neste caso, esta visão vai identificar os grupos que estão ativos no momento.

44

**Tabela 30-15. Colunas de `enabled_roles`**

Nome	Tipo de dado	Descrição
role_name	sql_identifier	Nome do grupo

**Exemplo:** Consultar a visão `enabled_roles`.<sup>45</sup>

```
=> SELECT * FROM information_schema.enabled_roles;
```

```

role_name
-----
engenharia
arquitetura
(2 linhas)
```

### 30.18. key\_column\_usage

A visão `key_column_usage` identifica todas as colunas do banco de dados corrente restringidas por uma restrição de unicidade, chave primária, ou chave estrangeira. As restrições de verificação não são incluídas nesta visão. Somente são mostradas as colunas contidas nas tabelas que pertencem ao usuário corrente.<sup>46 47 48</sup>

**Tabela 30-16. Colunas de `key_column_usage`**

Nome	Tipo de dado	Descrição
constraint_catalog	sql_identifier	Nome do banco de dados que contém a restrição (sempre o banco de dados corrente)
constraint_schema	sql_identifier	Nome do esquema que contém a restrição
constraint_name	sql_identifier	Nome da restrição
table_catalog	sql_identifier	Nome do banco de dados contendo a tabela que contém a coluna que é restringida por alguma restrição (sempre o banco de dados corrente)

Nome	Tipo de dado	Descrição
table_schema	sql_identifier	Nome do esquema contendo a tabela que contém a coluna que é restringida por alguma restrição
table_name	sql_identifier	Nome da tabela que contém a coluna que é restringida por alguma restrição
column_name	sql_identifier	Nome da coluna que é restringida por alguma restrição
ordinal_position	cardinal_number	Posição ordinal da coluna dentro da chave de restrição (contada a partir de 1)

**Exemplo:** Consultar a visão `key_column_usage`.<sup>49</sup>

```
=> \x
```

Habilitada a exibição expandida.

```
=> SELECT * FROM information_schema.key_column_usage;
```

```
-[ LINHA 1 ]-----+-----
constraint_catalog | teste
constraint_schema | public
constraint_name    | tbl_cardinal_pkey
table_catalog      | teste
table_schema       | public
table_name         | tbl_cardinal
column_name        | cardinal
ordinal_position   | 1
```

## 30.19. parameters

A visão `parameters` contém informações sobre os parâmetros (argumentos) de todas as funções no banco de dados corrente. Somente são mostradas as funções que o usuário corrente pode acessar (seja por ser o dono ou por possuir algum privilégio).<sup>50 51</sup>

**Tabela 30-17. Colunas de `parameters`**

Nome	Tipo de dado	Descrição
specific_catalog	sql_identifier	Nome do banco de dados que contém a função (sempre o banco de dados corrente)
specific_schema	sql_identifier	Nome do esquema que contém a função
specific_name	sql_identifier	O “nome específico” da função. Para obter informações adicionais deve ser consultada a Seção 30.26.
ordinal_position	cardinal_number	Posição ordinal do parâmetro na lista de argumentos da função (contada a partir de 1)
parameter_mode	character_data	Sempre <code>IN</code> , indicando um parâmetro de entrada (Futuramente poderão haver parâmetros com outros modos).
is_result	character_data	Se aplica a uma funcionalidade não disponível no PostgreSQL
as_locator	character_data	Se aplica a uma funcionalidade não disponível no PostgreSQL
parameter_name	sql_identifier	Nome do parâmetro, ou nulo se o parâmetro não possuir nome
data_type	character_data	Tipo de dado do parâmetro, se for um tipo nativo, ou <code>ARRAY</code> se for uma matriz (neste caso deve ser consultada a visão <code>element_types</code> ), ou então <code>USER-DEFINED</code> (neste caso, o

Nome	Tipo de dado	Descrição
		tipo é identificado em <code>udt_name</code> e nas colunas associadas).
<code>character_maximum_length</code>	<code>cardinal_number</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de parâmetro no PostgreSQL
<code>character_octet_length</code>	<code>cardinal_number</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de parâmetro no PostgreSQL
<code>character_set_catalog</code>	<code>sql_identifier</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>character_set_schema</code>	<code>sql_identifier</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>character_set_name</code>	<code>sql_identifier</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>collation_catalog</code>	<code>sql_identifier</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>collation_schema</code>	<code>sql_identifier</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>collation_name</code>	<code>sql_identifier</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>numeric_precision</code>	<code>cardinal_number</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de parâmetro no PostgreSQL
<code>numeric_precision_radix</code>	<code>cardinal_number</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de parâmetro no PostgreSQL
<code>numeric_scale</code>	<code>cardinal_number</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de parâmetro no PostgreSQL
<code>datetime_precision</code>	<code>cardinal_number</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de parâmetro no PostgreSQL
<code>interval_type</code>	<code>character_data</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de parâmetro no PostgreSQL
<code>interval_precision</code>	<code>character_data</code>	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado de parâmetro no PostgreSQL
<code>udt_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados onde o tipo de dado do parâmetro está definido (sempre o banco de dados corrente)
<code>udt_schema</code>	<code>sql_identifier</code>	Nome do esquema onde o tipo de dado do parâmetro está definido
<code>udt_name</code>	<code>sql_identifier</code>	Nome do tipo de dado do parâmetro
<code>scope_catalog</code>	<code>sql_identifier</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>scope_schema</code>	<code>sql_identifier</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>scope_name</code>	<code>sql_identifier</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>maximum_cardinality</code>	<code>cardinal_number</code>	Sempre nulo, porque as matrizes sempre possuem uma cardinalidade máxima não definida no PostgreSQL
<code>dtd_identifier</code>	<code>sql_identifier</code>	O identificador do descritor do tipo de dado do parâmetro, único entre os descritores de tipo de dado pertencentes à função. O uso principal é para fazer junção com outras instâncias de identificadores deste tipo (O formato específico do identificador não é definido, e também não há garantia que permaneça o mesmo nas versões futuras).



## 30.20. referential\_constraints

A visão `referential_constraints` contém todas as restrições referenciais (chaves estrangeiras) no banco de dados corrente pertencentes a uma tabela que pertence ao usuário corrente.<sup>52 53 54</sup>

**Tabela 30-18. Colunas de `referential_constraints`**

Nome	Tipo de dado	Descrição
<code>constraint_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a restrição (sempre o banco de dados corrente)
<code>constraint_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a restrição
<code>constraint_name</code>	<code>sql_identifier</code>	Nome da restrição
<code>unique_constraint_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a restrição de unicidade ou de chave primária, que a restrição de chave estrangeira faz referência (sempre o banco de dados corrente)
<code>unique_constraint_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a restrição de unicidade ou de chave primária que a restrição de chave estrangeira faz referência
<code>unique_constraint_name</code>	<code>sql_identifier</code>	Nome da restrição de unicidade ou de chave primária que a restrição de chave estrangeira faz referência
<code>match_option</code>	<code>character_data</code>	Opção de correspondência da restrição de chave estrangeira: FULL, PARTIAL ou NONE.
<code>update_rule</code>	<code>character_data</code>	Regra de atualização da restrição de chave estrangeira: CASCADE, SET NULL, SET DEFAULT, RESTRICT ou NO ACTION.
<code>delete_rule</code>	<code>character_data</code>	Regra de exclusão da restrição de chave estrangeira: CASCADE, SET NULL, SET DEFAULT, RESTRICT ou NO ACTION.

**Exemplo:** Consultar a visão `referential_constraints`.<sup>55</sup>

```
=> CREATE TABLE tbl_info (
(>     valor_cardinal  CARDINAL_NUMBER PRIMARY KEY REFERENCES tbl_cardinal,
(>     valor_caractere  CHARACTER_DATA
(> ) WITHOUT OIDS;
```

NOTA: `CREATE TABLE / PRIMARY KEY` criará índice implícito "tbl\_info\_pkey" na tabela "tbl\_info"

```
CREATE TABLE
```

```
=> \x
```

Habilitada a exibição expandida.

```
=> SELECT * FROM information_schema.referential_constraints;
```

```

-[ LINHA 1 ]-----+-----
constraint_catalog      | teste
constraint_schema      | public
constraint_name         | tbl_info_valor_cardinal_fkey
unique_constraint_catalog | teste
unique_constraint_schema | public
unique_constraint_name   | tbl_cardinal_pkey
match_option            | NONE
update_rule             | NO ACTION
delete_rule             | NO ACTION

```

### 30.21. role\_column\_grants

A visão `role_column_grants` identifica todos os privilégios concedidos em colunas para um grupo do qual o usuário corrente é membro. Podem ser encontradas informações adicionais em `column_privileges`.<sup>56</sup>

**Tabela 30-19. Colunas de `role_column_grants`**

Nome	Tipo de dado	Descrição
grantor	sql_identifier	Nome do usuário que concedeu o privilégio
grantee	sql_identifier	Nome do grupo para o qual o privilégio foi concedido
table_catalog	sql_identifier	Nome do banco de dados contendo a tabela que contém a coluna (sempre o banco de dados corrente)
table_schema	sql_identifier	Nome do esquema contendo a tabela que contém a coluna
table_name	sql_identifier	Nome da tabela que contém a coluna
column_name	sql_identifier	Nome da coluna
privilege_type	character_data	Tipo do privilégio: SELECT, INSERT, UPDATE ou REFERENCES
is_grantable	character_data	YES se o privilégio pode ser concedido, NO caso contrário

### 30.22. role\_routine\_grants

A visão `role_routine_grants` identifica todos os privilégios concedidos em funções para um grupo do qual o usuário corrente é membro. Podem ser encontradas informações adicionais em `routine_privileges`.<sup>57</sup>

**Tabela 30-20. Colunas de `role_routine_grants`**

Nome	Tipo de dado	Descrição
grantor	sql_identifier	Nome do usuário que concedeu o privilégio
grantee	sql_identifier	Nome do grupo para o qual o privilégio foi concedido
specific_catalog	sql_identifier	Nome do banco de dados que contém a função (sempre o banco de dados corrente)
specific_schema	sql_identifier	Nome do esquema que contém a função
specific_name	sql_identifier	O “nome específico” da função. Para obter informações adicionais deve ser consultada a Seção 30.26.
routine_catalog	sql_identifier	Nome do banco de dados que contém a função (sempre o banco de dados corrente)
routine_schema	sql_identifier	Nome do esquema que contém a função
routine_name	sql_identifier	Nome da função (pode ser duplicado em caso de sobrecarga)

Nome	Tipo de dado	Descrição
privilege_type	character_data	Sempre EXECUTE (o único tipo de privilégio para funções)
is_grantable	character_data	YES se o privilégio pode ser concedido, NO caso contrário

### 30.23. role\_table\_grants

A visão `role_table_grants` identifica todos os privilégios concedidos em tabelas ou visões para um grupo do qual o usuário corrente é membro. Podem ser encontradas informações adicionais em `table_privileges`.<sup>58</sup>

**Tabela 30-21. Colunas de `role_table_grants`**

Nome	Tipo de dado	Descrição
grantor	sql_identifier	Nome do usuário que concedeu o privilégio
grantee	sql_identifier	Nome do grupo para o qual o privilégio foi concedido
table_catalog	sql_identifier	Nome do banco de dados que contém a tabela (sempre o banco de dados corrente)
table_schema	sql_identifier	Nome do esquema que contém a tabela
table_name	sql_identifier	Nome da tabela
privilege_type	character_data	Tipo do privilégio: SELECT, DELETE, INSERT, UPDATE, REFERENCES, RULE ou TRIGGER
is_grantable	character_data	YES se o privilégio pode ser concedido, NO caso contrário
with_hierarchy	character_data	Se aplica a uma funcionalidade não disponível no PostgreSQL

### 30.24. role\_usage\_grants

A visão `role_usage_grants` tem por finalidade identificar os privilégios `USAGE` concedidos a vários tipos de objetos para um grupo do qual o usuário corrente é membro. Atualmente no PostgreSQL somente se aplica aos domínios, e uma vez que os domínios não possuem privilégios reais no PostgreSQL, esta visão é vazia. Podem ser encontradas informações adicionais em `usage_privileges`. No futuro, esta visão vai conter informações úteis.<sup>59</sup>

**Tabela 30-22. Colunas de `role_usage_grants`**

Nome	Tipo de dado	Descrição
grantor	sql_identifier	No futuro, o nome do usuário que concedeu o privilégio
grantee	sql_identifier	No futuro, o nome do grupo para o qual o privilégio foi concedido
object_catalog	sql_identifier	Nome do banco de dados que contém o objeto (sempre o banco de dados corrente)
object_schema	sql_identifier	Nome do esquema que contém o objeto
object_name	sql_identifier	Nome do objeto
object_type	character_data	No futuro, o tipo do objeto
privilege_type	character_data	Sempre USAGE
is_grantable	character_data	YES se o privilégio pode ser concedido, NO caso contrário

## 30.25. routine\_privileges

A visão `routine_privileges` identifica todos os privilégios concedidos em funções para o usuário corrente ou pelo usuário corrente. Existe uma linha para cada combinação de função, quem concedeu e a quem foi concedido. Os privilégios concedidos aos grupos são identificados na visão `role_routine_grants`.<sup>60</sup>

**Tabela 30-23. Colunas de `routine_privileges`**

Nome	Tipo de dado	Descrição
<code>grantor</code>	<code>sql_identifier</code>	Nome do usuário que concedeu o privilégio
<code>grantee</code>	<code>sql_identifier</code>	Nome do usuário ou do grupo para o qual o privilégio foi concedido
<code>specific_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a função (sempre o banco de dados corrente)
<code>specific_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a função
<code>specific_name</code>	<code>sql_identifier</code>	O “nome específico” da função. Para obter informações adicionais deve ser consultada a Seção 30.26.
<code>routine_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a função (sempre o banco de dados corrente)
<code>routine_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a função
<code>routine_name</code>	<code>sql_identifier</code>	Nome da função (pode ser duplicado em caso de sobrecarga)
<code>privilege_type</code>	<code>character_data</code>	Sempre <code>EXECUTE</code> (o único tipo de privilégio para funções)
<code>is_grantable</code>	<code>character_data</code>	<code>YES</code> se o privilégio pode ser concedido, <code>NO</code> caso contrário

Deve ser observado que a coluna `grantee` não faz distinção entre usuários e grupos. Havendo usuários e grupos com o mesmo nome, infelizmente não há maneira de distingui-los. Possivelmente será proibido existir usuários e grupos com o mesmo nome em uma versão futura do PostgreSQL.

## 30.26. routines

A visão `routines` contém todas as funções no banco de dados corrente. Somente são mostradas as funções que o usuário corrente pode acessar (seja por ser o dono ou por possuir algum privilégio).<sup>61 62</sup>

**Tabela 30-24. Colunas de `routines`**

Nome	Tipo de dado	Descrição
<code>specific_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a função (sempre o banco de dados corrente)
<code>specific_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a função
<code>specific_name</code>	<code>sql_identifier</code>	O “nome específico” da função. É um nome que identifica unicamente a função no esquema, mesmo quando o nome verdadeiro da função é sobrecarregado. O formato do nome específico não é definido, devendo ser utilizado apenas para comparar com outras instâncias de nomes específicos de rotinas.
<code>routine_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a função (sempre o banco de dados corrente)
<code>routine_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a função
<code>routine_name</code>	<code>sql_identifier</code>	Nome da função (pode ser duplicado em caso de sobrecarga)

Nome	Tipo de dado	Descrição
routine_type	character_data	Sempre FUNCTION (No futuro poderão existir outros tipos de rotina).
module_catalog	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
module_schema	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
module_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
udt_catalog	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
udt_schema	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
udt_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
data_type	character_data	Tipo de dado retornado pela função, se for um tipo nativo, ou ARRAY se for uma matriz (neste caso deve ser vista a visão element_types), senão USER-DEFINED (neste caso o tipo é identificado em type_udt_name e nas colunas associadas).
character_maximum_length	cardinal_number	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado retornados no PostgreSQL
character_octet_length	cardinal_number	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado retornados no PostgreSQL
character_set_catalog	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
character_set_schema	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
character_set_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
collation_catalog	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
collation_schema	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
collation_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
numeric_precision	cardinal_number	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado retornados no PostgreSQL
numeric_precision_radix	cardinal_number	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado retornados no PostgreSQL
numeric_scale	cardinal_number	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado retornados no PostgreSQL
datetime_precision	cardinal_number	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado retornados no PostgreSQL
interval_type	character_data	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado retornados no PostgreSQL
interval_precision	character_data	Sempre nulo, uma vez que esta informação não se aplica a tipos de dado retornados no PostgreSQL
type_udt_catalog	sql_identifier	Nome do banco de dados onde o tipo de dado retornado pela função está definido (sempre o banco de dados corrente)
type_udt_schema	sql_identifier	Nome do esquema onde o tipo de dado retornado pela função está definido
type_udt_name	sql_identifier	Nome do tipo de dado retornado pela função
scope_catalog	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL

Nome	Tipo de dado	Descrição
scope_schema	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
scope_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
maximum_cardinality	cardinal_number	Sempre nulo, porque as matrizes sempre possuem uma cardinalidade máxima não definida no PostgreSQL
dtd_identifier	sql_identifier	O identificador do descritor do tipo de dado retornado pela função, único entre os descritores de tipo de dado que pertencem à função. A utilidade principal é para fazer junção com outras instâncias destes identificadores (O formato específico do identificador não é definido, e também não há garantia que permaneça o mesmo nas versões futuras). (dtd = descritor do tipo de dado - N. do T.)
routine_body	character_data	Se a função for uma função SQL então SQL, senão EXTERNAL.
routine_definition	character_data	O texto fonte da função (nulo se o usuário corrente não for o dono da função) (De acordo com o padrão SQL, esta coluna somente se aplica se routine_body for SQL, mas no PostgreSQL contém qualquer que seja o texto fonte especificado quando a função foi criada).
external_name	character_data	Se a função for uma função C, então o nome externo (símbolo de ligação) da função; senão nulo (Acaba sendo o mesmo valor mostrado em routine_definition).
external_language	character_data	A linguagem na qual a função foi escrita
parameter_style	character_data	Sempre GENERAL (O padrão SQL define outros estilos de parâmetro, que não estão disponíveis no PostgreSQL).
is_deterministic	character_data	Se a função for declarada como imutável (chamado de determinística no padrão SQL) então YES, senão NO (Não é possível consultar os demais níveis de volatilidade disponíveis no PostgreSQL através do esquema de informações).
sql_data_access	character_data	Sempre MODIFIES, significando que a função possivelmente modifica dados SQL. Esta informação não é útil para o PostgreSQL.
is_null_call	character_data	Se a função retorna nulo automaticamente quando um de seus argumentos é nulo então YES, senão NO.
sql_path	character_data	Se aplica a uma funcionalidade não disponível no PostgreSQL
schema_level_routine	character_data	Sempre YES (O oposto seria um método de um tipo definido pelo usuário, que é uma funcionalidade não disponível no PostgreSQL.)
max_dynamic_result_sets	cardinal_number	Se aplica a uma funcionalidade não disponível no PostgreSQL
is_user_defined_cast	character_data	Se aplica a uma funcionalidade não disponível no PostgreSQL
is_implicitly_invocable	character_data	Se aplica a uma funcionalidade não disponível no PostgreSQL
security_type	character_data	Se a função executa com os privilégios do usuário corrente, então INVOKER, se a função executa com os privilégios do usuário que a definiu então DEFINER.
to_sql_specific_catalog	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL

Nome	Tipo de dado	Descrição
to_sql_specific_schema	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
to_sql_specific_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
as_locator	character_data	Se aplica a uma funcionalidade não disponível no PostgreSQL

## 30.27. schemata

A visão `schemata` contém todos os esquemas no banco de dados corrente pertencentes ao usuário corrente.<sup>63 64 65</sup>

**Tabela 30-25. Colunas de `schemata`**

Nome	Tipo de dado	Descrição
catalog_name	sql_identifier	Nome do banco de dados que contém o esquema (sempre o banco de dados corrente)
schema_name	sql_identifier	Nome do esquema
schema_owner	sql_identifier	Nome do dono do esquema
default_character_set_catalog	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
default_character_set_schema	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
default_character_set_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
sql_path	character_data	Se aplica a uma funcionalidade não disponível no PostgreSQL

**Exemplo:** Consultar a visão `schemata`.<sup>66</sup>

```
=> \c templatel postgres
```

Conectado ao banco de dados "templatel" como usuário "postgres".

```
=# SELECT catalog_name, schema_name, schema_owner FROM information_schema.schemata;
```

```
catalog_name |      schema_name      | schema_owner
-----+-----+-----
templatel   | pg_toast               | postgres
templatel   | pg_temp_1              | postgres
templatel   | pg_catalog              | postgres
templatel   | public                  | postgres
templatel   | information_schema      | postgres
(5 linhas)
```

```
=# \dn
```

```
          Lista de esquemas
          Nome      | Dono
-----+-----
information_schema | postgres
pg_catalog          | postgres
pg_toast             | postgres
public              | postgres
(4 linhas)
```

### 30.28. sql\_features

A tabela `sql_features` contém informações sobre quais funcionalidades formais definidas no padrão SQL são suportadas pelo PostgreSQL. É a mesma informação presente no Apêndice D, onde também podem ser encontradas informações adicionais.<sup>67</sup>

**Tabela 30-26. Colunas de `sql_features`**

Nome	Tipo de dado	Descrição
<code>feature_id</code>	<code>character_data</code>	Cadeia de caracteres identificadora da funcionalidade
<code>feature_name</code>	<code>character_data</code>	Nome descritivo da funcionalidade
<code>sub_feature_id</code>	<code>character_data</code>	Cadeia de caracteres identificadora da subfuncionalidade, ou uma cadeia de caracteres com comprimento zero se não for uma subfuncionalidade
<code>sub_feature_name</code>	<code>character_data</code>	Nome descritivo da subfuncionalidade, ou uma cadeia de caracteres com comprimento zero se não for uma subfuncionalidade
<code>is_supported</code>	<code>character_data</code>	YES se a funcionalidade for inteiramente suportada pela versão corrente do PostgreSQL, NO caso contrário
<code>is_verified_by</code>	<code>character_data</code>	Sempre nulo, uma vez que o grupo de desenvolvimento do PostgreSQL não efetua testes formais de conformidade das funcionalidades
<code>comments</code>	<code>character_data</code>	Um comentário sobre o status do suporte à funcionalidade

### 30.29. sql\_implementation\_info

A tabela `sql_implementation_info` contém informações sobre vários aspectos que o padrão SQL deixa para serem definidos pela implementação. A finalidade principal destas informações é serem utilizadas no contexto da interface ODBC; provavelmente os usuários de outras interfaces vão encontrar pouca utilidade nestas informações. Por esta razão, os itens de informação da implementação individuais não estão descritos nesta tabela; se encontram na descrição da interface ODBC.<sup>68</sup>

**Tabela 30-27. Colunas de `sql_implementation_info`**

Nome	Tipo de dado	Descrição
<code>implementation_info_id</code>	<code>character_data</code>	Cadeia de caracteres identificadora da informação de implementação do item
<code>implementation_info_name</code>	<code>character_data</code>	Nome descritivo da informação de implementação do item
<code>integer_value</code>	<code>cardinal_number</code>	Valor da informação de implementação do item, ou nulo se o valor estiver contido na coluna <code>character_value</code>
<code>character_value</code>	<code>character_data</code>	Valor da informação de implementação do item, ou nulo se o valor estiver contido na coluna <code>integer_value</code>
<code>comments</code>	<code>character_data</code>	Um comentário pertencente à informação de implementação do item

### 30.30. sql\_languages

A tabela `sql_languages` contém uma linha para cada ligação com linguagem SQL suportada pelo PostgreSQL. O PostgreSQL suporta SQL direto e SQL incorporado à linguagem C; isto é tudo o que se pode saber a partir desta tabela.<sup>69 70</sup>



Tabela 30-28. Colunas de `sql_languages`

Nome	Tipo de dado	Descrição
<code>sql_language_source</code>	<code>character_data</code>	O nome de origem da definição da linguagem; sempre ISO 9075, ou seja, o padrão SQL
<code>sql_language_year</code>	<code>character_data</code>	O ano em que o padrão referenciado em <code>sql_language_source</code> foi aprovado; atualmente 2003
<code>sql_language_comformance</code>	<code>character_data</code>	O nível de conformidade com o padrão desta ligação com a linguagem. Para ISO 9075:2003 é sempre CORE.
<code>sql_language_integrity</code>	<code>character_data</code>	Sempre nulo (Este valor tem relevância para uma versão anterior do padrão SQL)
<code>sql_language_implementation</code>	<code>character_data</code>	Sempre nulo
<code>sql_language_binding_style</code>	<code>character_data</code>	O estilo de ligação da linguagem, <code>DIRECT</code> (direto) ou <code>EMBEDDED</code> (incorporado)
<code>sql_language_programming_language</code>	<code>character_data</code>	A linguagem de programação, se o estilo de ligação for <code>EMBEDDED</code> , senão nulo. O PostgreSQL somente suporta a linguagem C.

**Exemplo:** Consultar a visão `sql_languages`.<sup>71</sup>

```
=> \x
```

Habilitada a exibição expandida.

```
=> SELECT * FROM information_schema.sql_languages;
```

```
-[ LINHA 1 ]-----+-----
sql_language_source      | ISO 9075
sql_language_year        | 2003
sql_language_conformance | CORE
sql_language_integrity   |
sql_language_implementation |
sql_language_binding_style | DIRECT
sql_language_programming_language |
-[ LINHA 2 ]-----+-----
sql_language_source      | ISO 9075
sql_language_year        | 2003
sql_language_conformance | CORE
sql_language_integrity   |
sql_language_implementation |
sql_language_binding_style | EMBEDDED
sql_language_programming_language | C
```

### 30.31. `sql_packages`

A tabela `sql_packages` contém informações sobre quais pacotes de funcionalidades definidos no padrão SQL são suportados pelo PostgreSQL. Para obter informações adicionais sobre pacotes de funcionalidades deve ser consultado o Apêndice D.<sup>72</sup>

Tabela 30-29. Colunas de `sql_packages`

Nome	Tipo de dado	Descrição
<code>feature_id</code>	<code>character_data</code>	Cadeia de caracteres identificadora do pacote
<code>feature_name</code>	<code>character_data</code>	Nome descritivo do pacote
<code>is_supported</code>	<code>character_data</code>	YES se o pacote for inteiramente suportado pela versão corrente do PostgreSQL, NO caso contrário
<code>is_verified_by</code>	<code>character_data</code>	Sempre nulo, uma vez que o grupo de desenvolvimento do PostgreSQL não efetua testes formais de conformidade das funcionalidades
<code>comments</code>	<code>character_data</code>	Um comentário sobre o status do suporte ao pacote

### 30.32. `sql_sizing`

A tabela `sql_sizing` contém informações sobre vários limites de tamanho e valores máximos no PostgreSQL. A finalidade principal destas informações é serem utilizadas no contexto da interface ODBC; provavelmente os usuários de outras interfaces vão encontrar pouca utilidade nestas informações. Por esta razão, os itens de tamanhos individuais não são descritos nesta tabela; se encontram na descrição da interface ODBC.<sup>73</sup>

Tabela 30-30. Colunas de `sql_sizing`

Nome	Tipo de dado	Descrição
<code>sizing_id</code>	<code>cardinal_number</code>	O identificador do item de tamanho
<code>sizing_name</code>	<code>character_data</code>	Nome descritivo do item de tamanho
<code>supported_value</code>	<code>cardinal_number</code>	Valor do item de tamanho, ou 0 se o tamanho não for limitado ou não puder ser determinado, ou nulo se a funcionalidade para a qual o item de tamanho se aplica não é suportada
<code>comments</code>	<code>character_data</code>	Um comentário pertencente ao item de tamanho

**Exemplo:** Consultar a visão `sql_sizing`.<sup>74</sup>

```
=> SELECT * FROM information_schema.sql_sizing;
```

```

sizing_id |              sizing_name              | supported_value |
comments
-----+-----+-----+-----
97 | MAXIMUM COLUMNS IN GROUP BY          | 0 |
99 | MAXIMUM COLUMNS IN ORDER BY          | 0 |
100 | MAXIMUM COLUMNS IN SELECT            | 1664 |
101 | MAXIMUM COLUMNS IN TABLE            | 1600 |
1 | MAXIMUM CONCURRENT ACTIVITIES          | 0 |
0 | MAXIMUM DRIVER CONNECTIONS            |
20000 | MAXIMUM STATEMENT OCTETS              | 0 |
20001 | MAXIMUM STATEMENT OCTETS DATA         | 0 |
20002 | MAXIMUM STATEMENT OCTETS SCHEMA        | 0 |
106 | MAXIMUM TABLES IN SELECT             | 0 |
25000 | MAXIMUM CURRENT DEFAULT TRANSFORM GROUP LENGTH |
25001 | MAXIMUM CURRENT TRANSFORM GROUP LENGTH |
25002 | MAXIMUM CURRENT PATH LENGTH           | 0 |
25003 | MAXIMUM CURRENT ROLE LENGTH           |
34 | MAXIMUM CATALOG NAME LENGTH           | 63 | Pode ser menos,
dependendo do conjunto de caracteres.
30 | MAXIMUM COLUMN NAME LENGTH            | 63 | Pode ser menos,
dependendo do conjunto de caracteres.
```

31   MAXIMUM CURSOR NAME LENGTH		63   Pode ser menos,
dependendo do conjunto de caracteres.		
10005   MAXIMUM IDENTIFIER LENGTH		63   Pode ser menos,
dependendo do conjunto de caracteres.		
32   MAXIMUM SCHEMA NAME LENGTH		63   Pode ser menos,
dependendo do conjunto de caracteres.		
35   MAXIMUM TABLE NAME LENGTH		63   Pode ser menos,
dependendo do conjunto de caracteres.		
107   MAXIMUM USER NAME LENGTH		63   Pode ser menos,
dependendo do conjunto de caracteres.		
25004   MAXIMUM SESSION USER LENGTH		63   Pode ser menos,
dependendo do conjunto de caracteres.		
25005   MAXIMUM SYSTEM USER LENGTH		63   Pode ser menos,
dependendo do conjunto de caracteres.		

(23 linhas)

### 30.33. sql\_sizing\_profiles

A tabela `sql_sizing_profiles` contém informações sobre os valores `sql_sizing` requeridos por vários perfis do padrão SQL. O PostgreSQL não acompanha qualquer perfil do SQL, portanto esta tabela é vazia.<sup>75</sup>

**Tabela 30-31. Colunas de `sql_sizing_profiles`**

Nome	Tipo de dado	Descrição
<code>sizing_id</code>	<code>cardinal_number</code>	Identificador do item de tamanho
<code>sizing_name</code>	<code>character_data</code>	Nome descritivo do item de tamanho
<code>profile_id</code>	<code>character_data</code>	Cadeia de caracteres identificadora do perfil
<code>required_value</code>	<code>cardinal_number</code>	O valor requerido pelo perfil SQL para o item de tamanho, ou 0 se o perfil não coloca nenhum limite para o item de tamanho, ou nulo se o perfil não requer nenhuma das funcionalidades para as quais o item de tamanho se aplica
<code>comments</code>	<code>character_data</code>	Um comentário pertencente ao item de tamanho no perfil

### 30.34. table\_constraints

A visão `table_constraints` contém todas as restrições pertencentes às tabelas que pertencem ao usuário corrente.<sup>76 77</sup>

**Tabela 30-32. Colunas de `table_constraints`**

Nome	Tipo de dado	Descrição
<code>constraint_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a restrição (sempre o banco de dados corrente)
<code>constraint_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a restrição
<code>constraint_name</code>	<code>sql_identifier</code>	Nome da restrição
<code>table_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a tabela (sempre o banco de dados corrente)
<code>table_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a tabela
<code>table_name</code>	<code>sql_identifier</code>	Nome da tabela
<code>constraint_type</code>	<code>character_data</code>	Tipo da restrição: CHECK, FOREIGN KEY, PRIMARY KEY ou UNIQUE
<code>is_deferrable</code>	<code>character_data</code>	YES se a restrição for postergável, NO caso contrário

Nome	Tipo de dado	Descrição
initially_deferred	character_data	YES se a restrição for postergável e inicialmente postergada, NO caso contrário

**Exemplo:** Consultar a visão `table_constraints`.<sup>79</sup>

```
=> \x
```

Habilitada a exibição expandida.

```
=> SELECT * FROM information_schema.table_constraints;
```

```
-[ LINHA 1 ]-----+-----
constraint_catalog | teste
constraint_schema | public
constraint_name    | tbl_cardinal_pkey
table_catalog     | teste
table_schema      | public
table_name        | tbl_cardinal
constraint_type    | PRIMARY KEY
is_deferrable     | NO
initially_deferred | NO
-[ LINHA 2 ]-----+-----
constraint_catalog | teste
constraint_schema | public
constraint_name    | tbl_info_pkey
table_catalog     | teste
table_schema      | public
table_name        | tbl_info
constraint_type    | PRIMARY KEY
is_deferrable     | NO
initially_deferred | NO
-[ LINHA 3 ]-----+-----
constraint_catalog | teste
constraint_schema | public
constraint_name    | tbl_info_valor_cardinal_fkey
table_catalog     | teste
table_schema      | public
table_name        | tbl_info
constraint_type    | FOREIGN KEY
is_deferrable     | NO
initially_deferred | NO
```

### 30.35. table\_privileges

A visão `table_privileges` identifica todos os privilégios concedidos em tabelas ou visões para o usuário corrente ou pelo usuário corrente. Existe uma linha para cada combinação de tabela, quem concedeu e a quem foi concedido. Os privilégios concedidos aos grupos são identificados na visão `role_table_grants`<sup>80 81 82</sup>

**Tabela 30-33. Colunas de `table_privileges`**

Nome	Tipo de dado	Descrição
grantor	sql_identifier	Nome do usuário que concedeu o privilégio
grantee	sql_identifier	Nome do usuário ou do grupo para o qual o privilégio foi concedido
table_catalog	sql_identifier	Nome do banco de dados que contém a tabela (sempre o banco de dados corrente)
table_schema	sql_identifier	Nome do esquema que contém a tabela

Nome	Tipo de dado	Descrição
table_name	sql_identifier	Nome da tabela
privilege_type	character_data	Tipo do privilégio: SELECT, DELETE, INSERT, UPDATE, REFERENCES, RULE ou TRIGGER
is_grantable	character_data	YES se o privilégio pode ser concedido, NO caso contrário
with_hierarchy	character_data	Se aplica a uma funcionalidade não disponível no PostgreSQL

Deve ser observado que a coluna `grantee` não faz distinção entre usuários e grupos. Havendo usuários e grupos com o mesmo nome, infelizmente não há maneira de distingui-los. Possivelmente será proibido existir usuários e grupos com o mesmo nome em uma versão futura do PostgreSQL.

### 30.36. tables

A visão `tables` contém todas as tabelas e visões definidas no banco de dados corrente. Somente são mostradas as tabelas e visões que o usuário corrente pode acessar (seja por ser o dono ou por possuir algum privilégio).<sup>83 84 85</sup>

**Tabela 30-34. Colunas de `tables`**

Nome	Tipo de dado	Descrição
table_catalog	sql_identifier	Nome do banco de dados que contém a tabela (sempre o banco de dados corrente)
table_schema	sql_identifier	Nome do esquema que contém a tabela
table_name	sql_identifier	Nome da tabela
table_type	character_data	Tipo da tabela: <code>BASE TABLE</code> para uma tabela base persistente <sup>a</sup> (o tipo normal de tabela), <code>VIEW</code> para uma visão, ou <code>LOCAL TEMPORARY</code> para uma tabela temporária
self_referencing_column_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
reference_generation	character_data	Se aplica a uma funcionalidade não disponível no PostgreSQL
user_defined_type_catalog	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
user_defined_type_schema	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
user_defined_type_name	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
Notas: a. persistente — que permanece existindo indefinidamente, até ser destruído deliberadamente. Ações referenciais e em cascata são consideradas como deliberadas. Ações ligadas ao término da transação SQL ou da sessão SQL não são consideradas como deliberadas. (ISO-ANSI Working Draft) Framework (SQL/Framework), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)		

**Exemplo:** Consultar a visão `tables`.<sup>86</sup>

```
=> \x
```

Habilitada a exibição expandida.

```
=> SELECT * FROM information_schema.tables WHERE table_schema='public';
```

```

-[ LINHA 1 ]-----+-----
table_catalog      | teste
table_schema      | public
table_name        | tbl_info
table_type        | BASE TABLE
self_referencing_column_name |
reference_generation |
user_defined_type_catalog |
user_defined_type_schema |
user_defined_name  |
-[ LINHA 2 ]-----+-----
table_catalog      | teste
table_schema      | public
table_name        | tbl_cardinal
table_type        | BASE TABLE
self_referencing_column_name |
reference_generation |
user_defined_type_catalog |
user_defined_type_schema |
user_defined_name  |

```

### 30.37. triggers

A visão `triggers` contém todos os gatilhos definidos no banco de dados corrente que pertencem ao usuário corrente (O dono da tabela é o dono do gatilho).<sup>87</sup>

**Tabela 30-35. Colunas de triggers**

Nome	Tipo de dado	Descrição
<code>trigger_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém o gatilho (sempre o banco de dados corrente)
<code>trigger_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém o gatilho
<code>trigger_name</code>	<code>sql_identifier</code>	Nome do gatilho
<code>event_manipulation</code>	<code>character_data</code>	O evento que dispara o gatilho (INSERT, UPDATE ou DELETE)
<code>event_object_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a tabela onde o gatilho está definido (sempre o banco de dados corrente)
<code>event_object_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a tabela onde o gatilho está definido
<code>event_object_name</code>	<code>sql_identifier</code>	Nome da tabela onde o gatilho está definido
<code>action_order</code>	<code>cardinal_number</code>	Ainda não implementado
<code>action_condition</code>	<code>character_data</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL
<code>action_statement</code>	<code>character_data</code>	Declaração executada pelo gatilho (atualmente sempre EXECUTE PROCEDURE <i>função(...)</i> )
<code>action_orientation</code>	<code>character_data</code>	Identifica se o gatilho dispara uma vez para cada linha processada ou uma vez para cada declaração (ROW ou STATEMENT)

Nome	Tipo de dado	Descrição
condition_timing	character_data	Momento em que o gatilho dispara (BEFORE ou AFTER)
condition_reference_old_table	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL
condition_reference_new_table	sql_identifier	Se aplica a uma funcionalidade não disponível no PostgreSQL

Os gatilhos no PostgreSQL possuem duas incompatibilidades com o padrão SQL que afetam a representação no esquema de informações: Em primeiro lugar, no PostgreSQL os nomes dos gatilhos são locais às tabelas, em vez de objetos independentes no esquema. Portanto, podem haver nomes de gatilhos duplicados definidos em um mesmo esquema, desde que pertençam a tabelas diferentes (`trigger_catalog` e `trigger_schema` são, na verdade, valores que pertencem à tabela onde o gatilho está definido); Em segundo lugar, no PostgreSQL os gatilhos podem ser definidos para disparar em vários eventos (por exemplo, `ON INSERT OR UPDATE`), enquanto o padrão SQL só permite um. Se o gatilho for definido para disparar em vários eventos, isto é representado por várias linhas no esquema de informações, uma para cada tipo de evento. Como consequência destas incompatibilidades, a chave primária da visão `triggers` na verdade é (`trigger_catalog`, `trigger_schema`, `trigger_name`, `event_object_name`, `event_manipulation`), em vez de (`trigger_catalog`, `trigger_schema`, `trigger_name`), conforme especificado pelo padrão SQL. Apesar disso, se os gatilhos forem definidos em conformidade com o padrão SQL (nomes de gatilho únicos no esquema e somente um evento por gatilho), estas incompatibilidades não trarão consequências.

### 30.38. usage\_privileges

A visão `usage_privileges` tem por finalidade identificar os privilégios `USAGE` concedidos em vários tipos de objeto para o usuário corrente ou pelo usuário corrente. Atualmente, no PostgreSQL, se aplica somente aos domínios e, uma vez que os domínios não possuem privilégios reais no PostgreSQL, esta visão mostra os privilégios `USAGE` implícitos concedidos a `PUBLIC` para todos os domínios. No futuro esta visão poderá conter informações mais úteis.<sup>88 89</sup>

**Tabela 30-36. Colunas de `usage_privileges`**

Nome	Tipo de dado	Descrição
grantor	sql_identifier	Atualmente definido com o nome do dono do objeto
grantee	sql_identifier	Atualmente sempre <code>PUBLIC</code>
object_catalog	sql_identifier	Nome do banco de dados que contém o objeto (sempre o banco de dados corrente)
object_schema	sql_identifier	Nome do esquema que contém o objeto
object_name	sql_identifier	Nome do objeto
object_type	character_data	Atualmente sempre <code>DOMAIN</code>
privilege_type	character_data	Sempre <code>USAGE</code>
is_grantable	character_data	Atualmente sempre <code>NO</code>

### 30.39. view\_column\_usage

A visão `view_column_usage` identifica todas as colunas utilizadas na expressão de consulta da visão (a declaração `SELECT` que define a visão). A coluna somente é incluída quando o usuário corrente é o dono da tabela que contém a coluna.<sup>90 91 92</sup>

**Nota:** As colunas das tabelas do sistema não são incluídas. Isto será corrigido alguma hora.

**Tabela 30-37. Colunas de `view_column_usage`**

Nome	Tipo de dado	Descrição
<code>view_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a visão (sempre o banco de dados corrente)
<code>view_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a visão
<code>view_name</code>	<code>sql_identifier</code>	Nome da visão
<code>table_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados contendo a tabela que contém a coluna utilizada pela visão (sempre o banco de dados corrente)
<code>table_schema</code>	<code>sql_identifier</code>	Nome do esquema contendo a tabela que contém a coluna utilizada pela visão
<code>table_name</code>	<code>sql_identifier</code>	Nome da tabela que contém a coluna utilizada pela visão
<code>column_name</code>	<code>sql_identifier</code>	Nome da coluna utilizada pela visão

### 30.40. `view_table_usage`

A visão `view_table_usage` identifica todas as tabelas utilizadas na expressão de consulta da visão (a declaração `SELECT` que define a visão). A tabela somente é incluída quando o usuário corrente é o dono da tabela.<sup>93 94 95</sup>

**Nota:** As tabelas do sistema não são incluídas. Isto será corrigido alguma hora.

**Tabela 30-38. Colunas de `view_table_usage`**

Nome	Tipo de dado	Descrição
<code>view_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a visão (sempre o banco de dados corrente)
<code>view_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a visão
<code>view_name</code>	<code>sql_identifier</code>	Nome da visão
<code>table_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a tabela que é utilizada pela visão (sempre o banco de dados corrente)
<code>table_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a tabela que é utilizada pela visão
<code>table_name</code>	<code>sql_identifier</code>	Nome da tabela que é utilizada pela visão

### 30.41. `views`

A visão `views` contém todas visões definidas no banco de dados corrente. Somente são mostradas as visões que o usuário corrente pode acessar (seja por ser o dono ou por possuir algum privilégio).<sup>96 97 98</sup>

**Tabela 30-39. Colunas de `views`**

Nome	Tipo de dado	Descrição
<code>table_catalog</code>	<code>sql_identifier</code>	Nome do banco de dados que contém a visão (sempre o banco de dados corrente)
<code>table_schema</code>	<code>sql_identifier</code>	Nome do esquema que contém a visão
<code>table_name</code>	<code>sql_identifier</code>	Nome da visão
<code>view definition</code>	<code>character_data</code>	Expressão de consulta que define a visão (nulo se o usuário corrente não for o dono da visão)
<code>check_option</code>	<code>character_data</code>	Se aplica a uma funcionalidade não disponível no PostgreSQL



Nome	Tipo de dado	Descrição
is_updatable	character_data	Ainda não implementado
is_insertable_into	character_data	Ainda não implementado

**Exemplo:** Consultar a visão views.<sup>99</sup>

```
=> CREATE VIEW vis_cardinal AS
->     SELECT *
->         FROM tbl_cardinal, tbl_info
->         WHERE cardinal=valor_cardinal;
CREATE VIEW
```

```
=> \x
```

Habilitada a exibição expandida.

```
=> SELECT * FROM information_schema.views WHERE table_schema='public';
```

```
-[ LINHA 1 ]-----+-----
table_catalog      | teste
table_schema      | public
table_name        | vis_cardinal
view_definition    | SELECT tbl_cardinal.cardinal, tbl_info.valor_cardinal, tbl_info.valor_caractere\
                   |         FROM tbl_cardinal, tbl_info\
                   |         WHERE ((tbl_cardinal.cardinal)::integer = (tbl_info.valor_cardinal)::integer);
check_option      | NONE
is_updatable      |
is_insertable_into |
```

```
=> \x
```

Desabilitada a exibição expandida.

```
=> \d vis_cardinal
```

```
                Visão "public.vis_cardinal"
      Coluna      |      Tipo      | Modificadores
-----+-----+-----
cardinal          | cardinal_number |
valor_cardinal    | cardinal_number |
valor_caractere   | character_data  |
Definição da visão:
SELECT tbl_cardinal.cardinal, tbl_info.valor_cardinal, tbl_info.valor_caractere
FROM tbl_cardinal, tbl_info
WHERE tbl_cardinal.cardinal::integer = tbl_info.valor_cardinal::integer;
```

## Notas

1. Oracle — o Oracle não possui nenhuma das visões do esquema de informações, entretanto estas informações podem ser obtidas em outras visões de metadados. Em vez de TABLES é usado ALL\_TABLES, em vez de COLUMNS é usado ALL\_TAB\_COLUMNS, e em vez de VIEWS é usado ALL\_VIEWS. Oracle and Standard SQL ([http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/ap\\_standard\\_sql.htm#7518](http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/ap_standard_sql.htm#7518)) (N. do T.)
2. SQL Server — o Microsoft® SQL Server 2000 disponibiliza dois métodos para obter metadados: procedimentos armazenados do sistema e visões do esquema de informações. Estas visões fornecem uma visão interna, independente das tabelas do sistema, dos metadados do SQL Server. As visões do esquema de informações permitem que os aplicativos trabalhem de forma apropriada, mesmo quando são feitas mudanças significativas nas tabelas do sistema. As visões do esquema de informações incluídas no SQL Server estão em conformidade com as definições do padrão SQL-92 para o INFORMATION\_SCHEMA. SQL Server Books Online (N. do T.)

3. DB2 — Cada banco de dados inclui um conjunto de tabelas de catálogo do sistema que descrevem a estrutura lógica e física dos dados, um arquivo de configuração contendo os valores dos parâmetros definidos para o banco de dados, e um registro de recuperação com as transações em andamento e transações arquiváveis. IBM Globalization - Terminology (<http://www-306.ibm.com/software/globalization/terminology/qr.html>) (N. do T.)
4. CREATE DOMAIN CARDINAL\_NUMBER AS INTEGER CONSTRAINT CARDINAL\_NUMBER\_DOMAIN\_CHECK CHECK (VALUE >= 0 ); (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
5. CREATE DOMAIN CHARACTER\_DATA AS CHARACTER VARYING (ML) CHARACTER SET SQL\_TEXT; (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
6. CREATE DOMAIN SQL\_IDENTIFIER AS CHARACTER VARYING(L) CHARACTER SET SQL\_IDENTIFIER; (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
7. CREATE DOMAIN TIME\_STAMP AS TIMESTAMP(2) WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP(2); (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
8. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
9. INFORMATION\_SCHEMA.CATALOG\_NAME — tabela base — Identifica o catálogo que contém o Esquema de Informações. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
10. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
11. APPLICABLE\_ROLES — visão — Identifica os grupos aplicáveis para o usuário corrente. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
12. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
13. CHECK\_CONSTRAINTS — visão — Identifica as restrições de verificação definidas neste catálogo que pertencem a um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
14. O rowset CHECK\_CONSTRAINTS identifica as restrições de verificação definidas no catálogo pertencentes a um determinado usuário. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
15. A visão CHECK\_CONSTRAINTS contém uma linha para cada restrição CHECK no banco de dados corrente. Esta visão do esquema de informações retorna informações sobre os objetos que o usuário corrente possui permissões. A visão INFORMATION\_SCHEMA.CHECK\_CONSTRAINTS é baseada nas tabelas do sistema sysobjects e syscomments. SQL Server Books Online (N. do T.)
16. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
17. COLUMN\_DOMAIN\_USAGE — visão — Identifica as colunas definidas que são dependentes de um domínio definido neste catálogo, e que pertencem a um usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
18. O rowset COLUMN\_DOMAIN\_USAGE identifica as colunas definidas no catálogo dependentes de um domínio definido no catálogo pertencentes a um determinado usuário. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
19. A visão COLUMN\_DOMAIN\_USAGE contém uma linha para cada coluna, no banco de dados corrente, que possui um tipo de dado definido pelo usuário. Esta visão do esquema de informações retorna informações sobre os objetos que o usuário corrente possui permissões. A visão INFORMATION\_SCHEMA.COLUMN\_DOMAIN\_USAGE é baseada nas tabelas do sistema sysobjects, syscolumns e systypes. SQL Server Books Online (N. do T.)
20. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
21. COLUMN\_PRIVILEGES — visão — Identifica os privilégios nas colunas das tabelas, definidos neste catálogo, que estão disponíveis ou foram concedidos por um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)

22. O rowset `COLUMN_PRIVILEGES` identifica os privilégios em colunas de tabelas definidos no catálogo disponíveis para, ou concedidos por, um determinado usuário. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
23. A visão `COLUMN_PRIVILEGES` contém uma linha para cada coluna com privilégio concedido por, ou para, o usuário corrente no banco de dados corrente. A visão `INFORMATION_SCHEMA.COLUMN_PRIVILEGES` é baseada nas tabelas do sistema `sysprotects`, `sysobjects` e `syscolumns`. SQL Server Books Online (N. do T.)
24. `COLUMN_UDT_USAGE` — visão — Identifica as colunas definidas que são dependentes de um tipo definido pelo usuário neste catálogo e possuído por um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
25. `COLUMNS` — visão — Identifica as colunas das tabelas definidas neste catálogo acessíveis a um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
26. O rowset `COLUMNS` identifica as colunas das tabelas (incluindo as visões) definidas no catálogo acessíveis por um determinado usuário. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
27. A visão `COLUMNS` contém uma linha para cada coluna que pode ser acessada pelo usuário corrente no banco de dados corrente. A visão `INFORMATION_SCHEMA.COLUMNS` é baseada nas tabelas do sistema `sysobjects`, `spt_data_type_info`, `systypes`, `syscolumns`, `syscomments`, `sysconfigures` e `syscharsets`. SQL Server Books Online (N. do T.)
28. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
29. `CONSTRAINT_COLUMN_USAGE` — visão — identifica as colunas utilizadas por restrições referenciais, restrições de unicidade, restrições de verificação e asserções definidas neste catálogo e que pertencem a um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
30. O rowset `CONSTRAINT_COLUMN_USAGE` identifica as colunas utilizadas por restrições referenciais, restrições de unicidade, restrições de verificação e asserções definidas no catálogo pertencentes a um determinado usuário. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
31. A visão `CONSTRAINT_COLUMN_USAGE` contém uma linha para cada coluna, no banco de dados corrente, que possui uma restrição definida nela. Esta visão do esquema de informações retorna informações sobre os objetos que o usuário corrente possui permissões. A visão `INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE` é baseada nas tabelas do sistema `sysobjects`, `syscolumns` e `systypes`. SQL Server Books Online (N. do T.)
32. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
33. `CONSTRAINT_TABLE_USAGE` — visão — Identifica as tabelas utilizadas por restrições referenciais, restrições de unicidade, restrições de verificação e asserções definidas neste catálogo e que pertencem a um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
34. O rowset `CONSTRAINT_TABLE_USAGE` identifica as tabelas utilizadas por restrições referenciais, restrições de unicidade, restrições de verificação e asserções definidas no catálogo pertencentes a um determinado usuário. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
35. A visão `CONSTRAINT_TABLE_USAGE` contém uma linha para cada tabela, no banco de dados corrente, que possui uma restrição definida nela. Esta visão do esquema de informações retorna informações sobre os objetos que o usuário corrente possui permissões. A visão `INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE` é baseada na tabela do sistema `sysobjects`. SQL Server Books Online (N. do T.)
36. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
37. `DOMAIN_CONSTRAINTS` — visão — Identifica as restrições de domínio dos domínios neste catálogo acessíveis a um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)

38. A visão `DOMAIN_CONSTRAINTS` contém uma linha para cada tipo de dado definido por usuário, que pode ser acessado pelo usuário corrente no banco de dados corrente, com uma regra ligada ao mesmo. A visão `INFORMATION_SCHEMA.DOMAIN_CONSTRAINTS` é baseada nas tabelas do sistema `sysobjects` e `systypes`. SQL Server Books Online (N. do T.)
39. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
40. `DOMAINS` — visão — Identifica os domínios definidos neste catálogo acessíveis por um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
41. A visão `DOMAINS` contém uma linha para cada tipo de dado definido por usuário, que pode ser acessado pelo usuário corrente no banco de dados corrente. A visão `INFORMATION_SCHEMA.DOMAINS` é baseada nas tabelas do sistema `spt_data`, `type_info`, `systypes`, `syscomments`, `sysconfigures` e `syscharsets`. SQL Server Books Online (N. do T.)
42. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
43. `ELEMENT_TYPES` — visão — Identifica a coleção de tipos de elemento definida neste catálogo acessíveis a um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
44. `ENABLED_ROLES` — visão — Identifica os grupos habilitados para a sessão SQL corrente. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
45. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
46. `KEY_COLUMN_USAGE` — visão — Identifica as colunas definidas neste catálogo restringidas como chave e acessíveis por um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
47. O rowset `KEY_COLUMN_USAGE` identifica as colunas definidas no catálogo restringidas como chave por um determinado usuário. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
48. A visão `KEY_COLUMN_USAGE` contém uma linha para cada coluna no banco de dados corrente que é restringida por uma chave. Esta visão do esquema de informações retorna informações sobre objetos que o usuário corrente possui permissões. A visão `INFORMATION_SCHEMA.KEY_COLUMN_USAGE` é baseada nas tabelas do sistema `sysobjects`, `syscolumns`, `sysreferences`, `spt_values` e `sysindexes`. SQL Server Books Online (N. do T.)
49. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
50. `PARAMETERS` — visão — Identifica os parâmetros SQL das rotinas chamadas pelo SQL definidas neste catálogo e acessíveis a um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
51. A visão `PARAMETERS` contém uma linha para cada parâmetro de função ou de procedimento armazenado definido por usuário, que pode ser acessado pelo usuário corrente no banco de dados corrente. Para as funções, esta visão também retorna uma linha com informações sobre o valor retornado. A visão `INFORMATION_SCHEMA.PARAMETERS` é baseada nas tabelas do sistema `sysobjects` e `syscolumns`. SQL Server Books Online (N. do T.)
52. `REFERENTIAL_CONSTRAINTS` — visão — Identifica as restrições referenciais definidas nas tabelas neste catálogo acessíveis a um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
53. O rowset `REFERENTIAL_CONSTRAINTS` identifica as restrições referenciais definidas no catálogo pertencentes a um determinado usuário. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
54. A visão `REFERENTIAL_CONSTRAINTS` contém uma linha para cada restrição de chave estrangeira no banco de dados corrente. Esta visão do esquema de informações retorna informações sobre os objetos que o usuário corrente possui permissões. A visão `INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS` é baseada nas tabelas do sistema `sysreferences`, `sysindexes` e `sysobjects`. SQL Server Books Online (N. do T.)
55. Exemplo escrito pelo tradutor, não fazendo parte do manual original.

56. `ROLE_COLUMN_GRANTS` — visão — Identifica os privilégios nas colunas definidos neste catálogo disponíveis para, ou concedidos pelos, grupos correntemente habilitados. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
57. `ROLE_ROUTINE_GRANTS` — visão — Identifica os privilégios nas rotinas chamadas pelo SQL definidas neste catálogo disponíveis para, ou concedidos pelos, grupos correntemente habilitados. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
58. `ROLE_TABLE_GRANTS` — visão — Identifica os privilégios nas tabelas definidos neste catálogo disponíveis para, ou concedidos pelos, grupos correntemente aplicáveis. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
59. `ROLE_USAGE_GRANTS` — visão — Identifica os privilégios `USAGE` nos objetos definidos neste catálogo disponíveis para, ou concedidos pelos, grupos correntemente habilitados. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
60. `ROUTINE_PRIVILEGES` — visão — Identifica os privilégios em rotinas chamadas pelo SQL definidos neste catálogo disponíveis para, ou concedidos por, um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
61. `ROUTINES` — visão — Identifica as rotinas chamadas pelo SQL neste catálogo acessíveis a um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
62. A visão `ROUTINES` contém uma linha para cada procedimento armazenado e função que pode ser acessada pelo usuário corrente no banco de dados corrente. As colunas que descrevem o valor retornado se aplicam apenas às funções. Para os procedimentos armazenados estas colunas têm o valor `NULL`. A visão `INFORMATION_SCHEMA.ROUTINES` é baseada nas tabelas do sistema `sysobjects` e `syscolumns`. SQL Server Books Online (N. do T.)
63. `schemata` — visão — Identifica os esquemas no catálogo que pertencem a um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
64. O `rowsetSCHEMATA` identifica os esquemas pertencentes a um determinado usuário. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
65. A visão `SCHEMATA` contém uma linha para cada banco de dados com permissão para o usuário corrente. A visão `INFORMATION_SCHEMA.SCHEMATA` é baseada nas tabelas do sistema `sysdatabases`, `sysconfigures` e `syscharsets`. SQL Server Books Online (N. do T.)
66. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
67. `SQL_FEATURES` — visão — Lista as funcionalidades e subfuncionalidades deste padrão, e indica quais destas a implementação dá suporte. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
68. `SQL_IMPLEMENTATION_INFO` — visão — Lista os itens de informação da implementação do SQL definidos neste padrão e, para cada um destes, indica os valores suportados pela implementação do SQL. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
69. `SQL_LANGUAGES` — visão — Identifica os níveis de conformidade, opções e dialetos suportados pela implementação-SQL de processamento de dados definida neste catálogo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
70. O `rowset SQL_LANGUAGES` identifica os níveis de conformidade, opções e dialetos suportados pela implementação-SQL de processamento de dados definida no catálogo. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
71. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
72. `SQL_PACKAGES` — visão — Lista os pacotes deste padrão, e indica quais destes a implementação do SQL suporta. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)

73. `SQL_SIZING` — visão — Lista os itens de tamanho definidos neste padrão e, para cada um destes, indica o tamanho suportado pela implementação SQL. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
74. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
75. `SQL_SIZING_PROFILES` — visão — Lista os itens de tamanho definidos no padrão e, para cada um destes, indica o tamanho requerido por um ou mais perfis do padrão. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
76. `TABLE_CONSTRAINTS` — visão — Identifica as restrições de tabela definidas nas tabelas neste catálogo acessíveis a um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
77. O rowset `TABLE_CONSTRAINTS` identifica as restrições de tabela definidas no catálogo pertencentes a um determinado usuário. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
78. A visão `TABLE_CONSTRAINTS` contém uma linha para cada restrição de tabela no banco de dados corrente. Esta visão do esquema de informações retorna informações sobre os objetos que o usuário corrente possui permissões. A visão `INFORMATION_SCHEMA.TABLE_CONSTRAINTS` é baseada na tabela do sistema `sysobjects`. SQL Server Books Online (N. do T.)
79. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
80. `TABLE_PRIVILEGES` — visão — Identifica os privilégios nas tabelas definidos neste catálogo disponíveis para, ou concedidos por, um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
81. O rowset `TABLE_PRIVILEGES` identifica os privilégios em tabelas definidos no catálogo disponíveis para, ou concedidos por, um determinado usuário. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
82. A visão `TABLE_PRIVILEGES` contém uma linha para cada privilégio de tabela concedido para, ou pelo, usuário corrente no banco de dados corrente. A visão `INFORMATION_SCHEMA.TABLE_PRIVILEGES` é baseada nas tabelas do sistema `sysprotects` e `sysobjects`. SQL Server Books Online (N. do T.)
83. `TABLES` — visão — Identifica as tabelas definidas neste catálogo acessíveis por um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
84. O rowset `TABLES` identifica as tabelas (incluindo as visões) definidas no catálogo acessíveis por um determinado usuário. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
85. A visão `TABLES` contém uma linha para cada tabela do banco de dados corrente na qual o usuário corrente possui permissões. A visão `INFORMATION_SCHEMA.TABLES` é baseada na tabela do sistema `sysobjects`. SQL Server Books Online (N. do T.)
86. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
87. `TRIGGERS` — visão — Identifica os gatilhos definidos nas tabelas neste catálogo acessíveis por um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
88. `USAGE_PRIVILEGES` — visão — Identifica os privilégios `USAGE` em objetos definidos neste catálogo disponíveis para, ou concedidos por, um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
89. O rowset `USAGE_PRIVILEGES` identifica os privilégios `USAGE` em objetos definidos no catálogo disponíveis para, ou concedidos por, um determinado usuário. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
90. `VIEW_COLUMN_USAGE` — visão — Identifica as colunas que as visões definidas neste catálogo e pertencentes a um determinado usuário ou grupo são dependentes. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)

91. O rowset VIEW\_COLUMN\_USAGE identifica as colunas que as visões definidas no catálogo, pertencentes a um determinado usuário, são dependentes. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
92. A visão VIEW\_COLUMN\_USAGE contém uma linha para cada coluna, no banco de dados corrente, usada em uma definição de visão. Esta visão do esquema de informações retorna informações sobre os objetos que o usuário corrente possui permissões. A visão INFORMATION\_SCHEMA.VIEW\_COLUMN\_USAGE é baseada nas tabelas do sistema sysobjects e sysdepends. SQL Server Books Online (N. do T.)
93. VIEW\_TABLE\_USAGE — visão — Identifica as tabelas que as visões definidas neste catálogo pertencentes a um determinado usuário ou grupo são dependentes. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
94. O rowset VIEW\_TABLE\_USAGE identifica as tabelas que as visões definidas no catálogo, pertencentes a um determinado usuário, são dependentes. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
95. A visão VIEW\_TABLE\_USAGE contém uma linha para cada tabela, no banco de dados corrente, usada em uma visão. Esta visão do esquema de informações retorna informações sobre os objetos que o usuário corrente possui permissões. A visão INFORMATION\_SCHEMA.VIEW\_TABLE\_USAGE é baseada nas tabelas do sistema sysobjects e sysdepends. SQL Server Books Online (N. do T.)
96. VIEWS — visão — Identifica as visões definidas neste catálogo acessíveis por um determinado usuário ou grupo. (ISO-ANSI Working Draft) Information and Definition Schemas (SQL/Schemata), ISO/IEC 9075-11:2003 (E) (N. do T.)
97. O rowset VIEWS identifica as visões definidas no catálogo acessíveis por um determinado usuário. Microsoft OLE DB Programmer's Reference ([http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema\\_rowsets.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbschema_rowsets.asp)) (N. do T.)
98. A visão VIEWS contém uma linha para cada visão que pode ser acessada pelo usuário corrente no banco de dados corrente. A visão INFORMATION\_SCHEMA.VIEWS é baseada nas tabelas do sistema sysobjects e syscomments. SQL Server Books Online (N. do T.)
99. Exemplo escrito pelo tradutor, não fazendo parte do manual original.

## V. Programação servidor

Esta parte diz respeito à extensão das funcionalidades do servidor através de funções, tipos de dado, gatilhos, etc. definidos pelo usuário. Estes tópicos são avançados e, provavelmente, somente deverão ser estudados após as demais partes da documentação do PostgreSQL para usuários tiver sido compreendida. Os últimos capítulos desta parte descrevem as linguagens de programação do lado servidor disponíveis na distribuição do PostgreSQL, assim como questões gerais sobre as linguagens de programação do lado servidor. É essencial ler ao menos as primeiras seções do Capítulo 31 (cobrindo as funções), antes de se aprofundar no material sobre linguagens de programação do lado servidor.



# Capítulo 31. Estendendo a linguagem SQL

Nas próximas seções será mostrado como se pode estender a linguagem de comandos SQL do PostgreSQL pela adição de:

- funções (começando na Seção 31.3)
- agregações (começando na Seção 31.10)
- tipos de dado (começando na Seção 31.11)
- operadores (começando na Seção 31.12)
- classes de operador para índices (começando na Seção 31.14)

## 31.1. Como funciona a extensibilidade

O PostgreSQL é extensível, porque sua operação é dirigida pelo catálogo (*catalog-driven*). Quem está familiarizado com sistemas de banco de dados relacionais padrão com certeza sabe que estes armazenam informações sobre bancos de dados, tabelas, colunas, etc., no que é comumente conhecido por catálogos do sistema (Alguns sistemas chamam de dicionário de dados). Os catálogos são vistos pelo usuário como qualquer outra tabela, mas o SGBD armazena suas informações internas nestas tabelas. Uma diferença chave entre o PostgreSQL e os sistemas de banco de dados relacionais padrão é que o PostgreSQL armazena muito mais informação em seus catálogos: não apenas informações sobre tabelas e colunas, mas também informações sobre tipos de dado, funções, métodos de acesso, etc. Estas tabelas podem ser modificadas pelo usuário e, uma vez que as operações do PostgreSQL são baseadas nestas tabelas, isto significa que o PostgreSQL pode ser estendido pelos usuários. Em comparação, os sistemas de banco de dados convencionais só podem ser estendidos alterando os procedimentos presentes no código fonte, ou pela carga de módulos escritos pelo fornecedor do SGBD.

Além disso, o servidor PostgreSQL pode incorporar código escrito pelo usuário através de carregamento dinâmico, ou seja, o usuário especifica um arquivo contendo código objeto (por exemplo, uma biblioteca compartilhada implementando um novo tipo de dado ou função), e o PostgreSQL fará a carga deste módulo quando houver necessidade. O código escrito na linguagem SQL é ainda mais simples de ser adicionado ao servidor. Esta capacidade de modificar sua operação “em tempo de execução” (*on the fly*) torna o PostgreSQL especialmente indicado para a prototipação rápida de novos aplicativos e estruturas de armazenamento.

## 31.2. O sistema de tipos de dado do PostgreSQL

Os tipos de dado do PostgreSQL são divididos em tipos base, tipos compostos, domínios e pseudotipos.

### 31.2.1. Tipos base

Os tipos base, como o `int4`, são aqueles implementados abaixo do nível da linguagem SQL (tipicamente em uma linguagem de baixo nível, como a linguagem C). Correspondem, geralmente, ao que normalmente é conhecido como tipo de dado abstrato (ADT).<sup>1</sup> O PostgreSQL somente opera sobre estes tipos de dado através das funções fornecidas pelo usuário, e somente compreende o comportamento destes tipos de dado no grau em que são descritos pelo usuário. Por sua vez, os tipos de dado base são subdivididos em escalar e matriz. Para cada tipo escalar é criado, automaticamente, um tipo matriz correspondente podendo conter matrizes de tamanho variável deste tipo escalar.

### 31.2.2. Tipos compostos

Os tipos compostos, ou tipos linha, são criados toda vez que o usuário cria uma tabela; também é possível definir um tipo composto “autônomo”, sem nenhuma tabela associada. Um tipo composto é simplesmente uma lista de tipos base com nomes de campo associados. O valor de um tipo composto é uma linha ou registro de valores de campo. O usuário pode acessar os campos componentes a partir de consultas SQL.

### 31.2.3. Domínios

Um domínio se baseia em um determinado tipo base e, para muitas finalidades, é intercambiável com o seu tipo base. Entretanto, o domínio pode ter restrições limitando os valores válidos a um subconjunto dos valores permitidos pelo tipo base subjacente.

Os domínios podem ser criados utilizando o comando *CREATE DOMAIN* do SQL. A criação e uso de domínios não são vistos neste capítulo.

### 31.2.4. Pseudotipos

Existem alguns poucos “pseudotipos” para finalidades especiais. Os pseudotipos não podem aparecer como colunas de tabela ou atributos de tipo composto, mas podem ser usados para declarar tipos de dado de argumentos e resultados de funções. Fornecem um mecanismo dentro do sistema de tipos para identificar classes especiais de funções. A Tabela 8-20 lista os pseudotipos existentes.

### 31.2.5. Tipos polimórficos

Dois pseudotipos de especial interesse são *anyelement* e *anyarray*, chamados coletivamente de *tipos polimórficos*. Qualquer função declarada utilizando um destes tipos é dita como sendo uma *função polimórfica*. Uma função polimórfica pode operar sobre muitos tipos de dado diferentes, sendo o tipo de dado específico determinado pelo tipo de dado passado para a função na hora da chamada.

Os argumentos e resultados polimórficos estão presos um ao outro, sendo determinados como um tipo de dado específico quando o comando que faz a chamada à função polimórfica é analisado. Todas as posições (tanto argumento como valor retornado) declaradas como *anyelement* podem receber qualquer tipo de dado, mas em uma determinada chamada todas devem ter o *mesmo* tipo. Todas as posições declaradas como *anyarray* podem ter qualquer tipo de dado matriz, mas de forma análoga todas as posições devem ser do mesmo tipo. Havendo algumas posições declaradas como *anyarray* e outras declaradas como *anyelement*, o tipo matriz nas posições *anyarray* devem ser matrizes cujos elementos são do mesmo tipo aparecendo nas posições *anyelement*.

Portanto, quando são declarados argumentos de tipo polimórfico em mais de uma posição, resulta em que apenas certas combinações de tipos de argumento são permitidas. Por exemplo, uma função declarada como `igual(anyelement, anyelement)` recebe quaisquer dois valores de entrada, desde que ambos sejam do mesmo tipo de dado.

Quando o valor retornado por uma função é declarado como sendo do tipo polimórfico, deve haver pelo menos uma posição de argumento que também seja polimórfica, e o tipo de dado fornecido como argumento determina o tipo de dado real do resultado para a chamada. Por exemplo, se já não houvesse um mecanismo de índice para matriz, este poderia ser implementado por uma função como `indice(anyarray, integer) returns anyelement`. Esta declaração restringe o primeiro argumento como sendo do tipo matriz, permitindo ao analisador inferir o tipo de dado correto do resultado a partir do tipo de dado do primeiro argumento.

## 31.3. Funções definidas pelo usuário

O PostgreSQL possui quatro tipos de função:

- funções na linguagem de comando (funções escritas em SQL) (Seção 31.4)
- funções nas linguagens procedurais (funções escritas em, por exemplo, PL/pgSQL ou PL/Tcl) (Seção 31.7)
- funções internas (Seção 31.8)
- funções na linguagem C (Seção 31.9)

Todos os tipos de função aceitam tipos base, tipos compostos, ou alguma combinação destes tipos como argumentos (parâmetros). Além disso, todos os tipos de função podem retornar um tipo base ou um tipo composto. As funções também podem ser definidas como retornando um conjunto de valores base ou compostos.

Vários tipos de função podem receber ou retornar certos pseudotipos (tal como os tipos polimórficos), mas as funcionalidades disponíveis podem variar. Para obter mais detalhes deve ser consultada a descrição de cada tipo de função.

As funções SQL são as mais fáceis de serem definidas e, portanto, começaremos por estas. A maior parte dos conceitos apresentados para as funções SQL podem ser levados para os outros tipos de função.

Durante a leitura deste capítulo pode ser útil consultar a página de referência do comando *CREATE FUNCTION* para compreender melhor os exemplos. Alguns exemplos deste capítulo podem ser encontrados nos arquivos `funcs.sql` e `funcs.c` na distribuição do código fonte do PostgreSQL, no diretório `src/tutorial`.

## 31.4. Funções na linguagem de comando (SQL)

As funções SQL executam uma lista arbitrária de declarações SQL, retornando o resultado da última consulta da lista. No caso mais simples (não-conjunto), a primeira linha do resultado da última consulta é retornada (Deve-se ter em mente que a “primeira linha” de um resultado de várias linhas não é bem definido, a menos que seja utilizada a cláusula `ORDER BY`). Caso a última consulta não retorne nenhuma linha, é retornado o valor nulo.

Como alternativa a função SQL pode ser declarada como retornando um conjunto, especificando o tipo retornado pela função como `SETOF algum_tipo`. Neste caso, todas as linhas do resultado da última consulta são retornadas. Abaixo são mostrados mais detalhes.

O corpo de uma função SQL deve ser uma lista contendo uma ou mais declarações SQL separadas por ponto-e-vírgula (;). O ponto-e-vírgula após a última declaração é opcional. A menos que a função seja declarada como retornando o tipo `void`, a última declaração deve ser um comando `SELECT`.

Qualquer coleção de comandos na linguagem SQL pode ser juntado e definido como uma função. Além de comandos `SELECT`, podem existir comandos de manipulação de dados (`INSERT`, `UPDATE` e `DELETE`), assim como outros comandos SQL (A única exceção é que não se pode colocar os comandos `BEGIN`, `COMMIT`, `ROLLBACK` ou `SAVEPOINT` na função SQL). Entretanto, o comando final deve ser um `SELECT` retornando o que foi especificado como sendo o tipo retornado pela função. Como alternativa, se for desejado definir uma função SQL que realiza ações mas não retorna um valor útil, a função pode ser definida como retornando `void`. Neste caso, o corpo da função não deve terminar por um comando `SELECT`. Por exemplo, a função abaixo remove as linhas contendo salários negativos da tabela `emp`:

```
CREATE FUNCTION limpar_emp () RETURNS void AS '
    DELETE FROM emp
        WHERE emp.salario <= 0;
' LANGUAGE SQL;
```

```
SELECT limpar_emp();
```

```
limpar_emp
-----
```

```
(1 linha)
```

A sintaxe do comando `CREATE FUNCTION` requer que o corpo da função seja escrito como uma constante cadeia de caracteres. Geralmente é mais fácil utilizar a notação de cifrão (\$) (consulte a Seção 4.1.2.2) para a constante cadeia de caracteres. Se for utilizada a sintaxe regular de constante cadeia de caracteres que utiliza apóstrofes, os apóstrofes (') e as contrabarras (\) presentes no corpo da função devem ser precedidos por caractere de escape, tipicamente duplicando estes caracteres (consulte a Seção 4.1.2.1).

Os argumentos da função SQL são referenciados no corpo da função utilizando a sintaxe `$n`: `$1` se refere ao primeiro argumento, `$2` ao segundo, e assim por diante. Se o argumento for de um tipo composto, então pode ser utilizada a “notação de ponto”, por exemplo `$1.nome`, para acessar os atributos do argumento. Os argumentos só podem ser utilizados como valores de dado, e não como identificadores. Portanto, por exemplo, isto faz sentido:

```
INSERT INTO minha_tabela VALUES ($1);
```

mas isto não funciona:

```
INSERT INTO $1 VALUES (42);
```

### 31.4.1. Funções SQL com tipos base

O tipo mais simples possível de função SQL não possui argumentos, e simplesmente retorna um tipo base como o `integer`:

```
CREATE FUNCTION um() RETURNS integer AS $$
    SELECT 1 AS resultado;
$$ LANGUAGE SQL;
```

```
-- Sintaxe alternativa para o literal cadeia de caracteres:
```

```
CREATE FUNCTION um() RETURNS integer AS '
    SELECT 1 AS resultado;
' LANGUAGE SQL;
```

```
SELECT um();
```

```
um
----
1
(1 linha)
```

Deve ser observado que foi definido um aliás de coluna dentro do corpo da função para o resultado da função (com o nome `resultado`), mas este aliás de coluna não é visível fora da função. Portanto, o rótulo do resultado é `um` em vez de `resultado`.

Definir funções SQL que recebem tipos base como argumentos é praticamente tão simples quanto este último exemplo. No exemplo abaixo deve ser observado que, dentro da função, os argumentos são referenciados como `$1` e `$2`:

```
CREATE FUNCTION somar(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;
```

```
SELECT somar(1, 2) AS resposta;
```

```
resposta
-----
3
```

Abaixo está mostrada uma função mais útil, que pode ser utilizada para realizar débitos em uma conta corrente no banco:

```
CREATE FUNCTION debitar (integer, numeric) RETURNS integer AS $$
    UPDATE conta_corrente
        SET saldo = saldo - $2
        WHERE numero_da_conta = $1;
    SELECT 1;
$$ LANGUAGE SQL;
```

O usuário pode executar esta função para debitar R\$100.00 da conta 17 da seguinte maneira:

```
SELECT debitar(17, 100.0);
```

Provavelmente, na prática seria desejado que a função retornasse um resultado mais útil do que a constante “1” e, portanto, uma definição mais realística seria

```
CREATE FUNCTION debitar (integer, numeric) RETURNS numeric AS $$
    UPDATE conta_corrente
        SET saldo = saldo - $2
        WHERE numero_da_conta = $1;
    SELECT saldo FROM conta_corrente WHERE numero_da_conta = $1;
$$ LANGUAGE SQL;
```

que atualiza o saldo e retorna o novo saldo.

### 31.4.2. Funções SQL com tipos compostos

Ao escrever funções com argumentos de tipo composto, não se deve especificar apenas qual é o argumento desejado (como foi feito acima com `$1` e `$2`), mas também qual é o atributo (campo) do argumento desejado. Por exemplo, supondo que `emp` seja uma tabela contendo dados dos empregados e, portanto, também o nome do tipo composto de cada linha da tabela, a função `dobrar_salario` mostrada abaixo calcula qual seria o salário de alguém caso este fosse dobrado:

```

CREATE TABLE emp (
    nome          text,
    salario       numeric,
    idade         integer,
    baia          point
);

INSERT INTO emp VALUES('João',2200,21,point('(1,1)'));
INSERT INTO emp VALUES('José',4200,30,point('(2,1)'));

CREATE FUNCTION dobrar_salario(emp) RETURNS numeric AS $$
    SELECT $1.salario * 2 AS salario;
$$ LANGUAGE SQL;

SELECT nome, dobrar_salario(emp.*) AS sonho
    FROM emp
    WHERE emp.baia ~= point '(2,1)';

```

```

nome | sonho
-----+-----
José |  8400
(1 linha)

```

Deve ser observada a utilização da sintaxe `$1.salario` para especificar o campo da linha passada como argumento. Deve ser observado, também, a utilização no comando `SELECT` do `*` para selecionar toda a linha corrente da tabela como um valor composto. Como alternativa, a linha da tabela pode ser referenciada usando apenas o nome da tabela, como mostrado abaixo:

```

SELECT nome, dobrar_salario(emp) AS sonho
    FROM emp
    WHERE emp.baia ~= point '(2,1)';

```

mas esta forma de utilização está em obsolescência, uma vez que é propensa a causar confusão.

Algumas vezes é prático gerar o valor do argumento composto em tempo de execução. Isto pode ser feito através da construção `ROW`. Por exemplo, os dados passados para esta função poderiam estar na forma:

```

SELECT nome, dobrar_salario(ROW(nome, salario*1.1, idade, baia)) AS sonho
    FROM emp;

```

```

nome | sonho
-----+-----
João | 4840.0
José | 9240.0
(2 linhas)

```

Também é possível construir uma função que retorna um tipo composto. Abaixo está mostrado como exemplo uma função que retorna uma única linha da tabela `emp`:

```

CREATE FUNCTION novo_empregado() RETURNS emp AS $$
    SELECT text 'Nenhum' AS nome,
        1000.0 AS salario,
        25 AS idade,
        point '(2,2)' AS baia;
$$ LANGUAGE SQL;

```

Neste exemplo cada um dos atributos foi especificado através de um valor constante, mas estas constantes poderiam ser substituídas por algum valor calculado.

Devem ser observados dois fatos importantes sobre a definição da função:

- A ordem da lista de seleção da consulta deve ser exatamente a mesma em que as colunas aparecem na tabela associada ao tipo composto (Dar nome às colunas, como foi feito acima, é irrelevante para o sistema).

- Deve ser feita a conversão de tipo nas expressões para haver correspondência com a definição do tipo composto, ou acontecerá um erro deste tipo:

```
ERRO: função declarada como retornando emp retorna varchar em vez de text para a coluna 1
```

Uma forma diferente para definir a mesma função seria:

```
CREATE OR REPLACE FUNCTION novo_empregado() RETURNS emp AS $$
    SELECT ROW('Nenhum', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

Neste caso foi escrito um comando `SELECT` que retorna apenas uma única coluna do tipo composto correto. Não é realmente o melhor nesta situação, mas é uma alternativa prática em algumas situações — por exemplo, se for necessário calcular o resultado chamando outra função que recebe como argumento o valor composto retornado por esta função.

Esta função poderia ser chamada diretamente de uma destas duas maneiras:

```
SELECT novo_empregado();
```

```

      novo_empregado
-----
(Nenhum,1000.0,25,"(2,2)")
(1 linha)
```

```
SELECT * FROM novo_empregado();
```

```

 nome | salario | idade | baia
-----+-----+-----+-----
Nenhum | 1000.0 | 25 | (2,2)
(1 linha)
```

A segunda forma está descrita com mais detalhes na Seção 31.4.3.

Quando se usa uma função que retorna um tipo composto, pode ser desejado apenas um campo (atributo) de seu resultado. Isto pode ser feito utilizando uma sintaxe do tipo:

```
SELECT (novo_empregado()).nome;
```

```

      nome
-----
Nenhum
(1 linha)
```

Os parênteses adicionais são necessários para evitar que o analisador se confunda. Se for tentado fazer a consulta sem usar os parênteses adicionais será recebida uma mensagem de erro como esta:

```
SELECT novo_empregado().nome;
ERRO: erro de sintaxe em ou próximo a "." no caractere 24
LINHA 1: SELECT novo_empregado().nome;
```

Outra opção é utilizar a notação de função para extrair o atributo. A forma mais fácil de explicar isto é dizer que pode ser utilizada tanto a notação `atributo(tabela)` quanto a notação `tabela.atributo`, indiferentemente:

```
SELECT nome(novo_empregado());
```

```

      nome
-----
Nenhum
(1 linha)
```

```
SELECT emp.nome AS jovens FROM emp WHERE emp.idade < 30;
```

```
-- é o mesmo que:
```

```
SELECT nome(emp) AS jovens FROM emp WHERE idade(emp) < 30;
```

```
jovens
-----
João
(1 linha)
```

**Dica:** A equivalência entre a notação de função e a notação de atributo torna possível utilizar funções com tipos compostos para emular “campos calculados”. Por exemplo, utilizando a definição anterior para `dobrar_salario(emp)` pode ser escrito:

```
SELECT emp.nome, emp.dobrar_salario FROM emp;
```

```
nome | dobrar_salario
-----+-----
João |           4400
José |           8400
(2 linhas)
```

Um aplicativo utilizando esta sintaxe não necessita se preocupar diretamente com o fato de que `dobrar_salario` não é uma coluna real da tabela (Os campos calculados também podem ser emulados através de funções).

Outra maneira de utilizar uma função que retorna uma linha como resultado é passando o resultado desta função para outra função que recebe este tipo de linha como argumento:

```
CREATE FUNCTION obtem_nome(emp) RETURNS text AS $$
    SELECT $1.nome;
$$ LANGUAGE SQL;
```

```
SELECT obtem_nome(novo_empregado());
```

```
obtem_nome
-----
Nenhum
(1 linha)
```

Uma outra maneira de utilizar uma função que retorna um tipo composto é chamá-la como uma função de tabela, conforme descrito abaixo.

### 31.4.3. Funções SQL como fontes de tabela

Todas as funções SQL podem ser utilizadas na cláusula `FROM` da consulta, mas esta situação é particularmente útil no caso das funções que retornam tipos compostos. Se a função for definida como retornando um tipo base, a função de tabela produz uma tabela de uma coluna. Se a função for definida como retornando um tipo composto, a função de tabela produz uma coluna para cada atributo do tipo composto.

Abaixo segue um exemplo:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'João');
INSERT INTO foo VALUES (1, 2, 'José');
INSERT INTO foo VALUES (2, 1, 'Maria');
```

```
CREATE FUNCTION getfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;
```

```
SELECT *, upper(fooname) FROM getfoo(1) AS t1;
```

```
fooid | foosubid | fooname | upper
-----+-----+-----+-----
1 | 1 | João | JOÃO
(1 linha)
```

Conforme mostrado neste exemplo, as colunas do resultado da função podem ser utilizadas da mesma maneira como se fossem colunas de uma tabela comum.

Deve ser observado que a função somente retornou uma linha. Isto se deve a não utilização de `SETOF`, que será descrito na próxima seção.

#### 31.4.4. Funções SQL que retornam conjunto

Quando uma função SQL é declarada como retornando `SETOF algum_tipo`, a consulta `SELECT` no final da função é executada até o fim, e cada linha produzida é retornada como um elemento do conjunto resultado.

Esta funcionalidade normalmente é utilizada quando se chama a função na cláusula `FROM`. Neste caso, cada linha retornada pela função se torna uma linha da tabela vista pela consulta. Por exemplo, assumindo que a tabela `foo` possui o mesmo conteúdo mostrado acima, então:

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

Resultaria em:

```
fooid | foosubid | fooname
-----+-----+-----
      1 |         1 | João
      1 |         2 | José
(2 linhas)
```

Atualmente as funções que retornam conjunto também podem ser chamadas na lista de seleção da consulta. Para cada linha que a consulta gera por si própria é chamada a função que retorna conjunto, sendo gerada uma linha de saída para cada elemento do conjunto resultado da função. Entretanto, deve ser observado que esta funcionalidade está em obsolescência (*deprecated*), podendo ser removida em uma versão futura. Abaixo está mostrada, como exemplo, uma função retornando um conjunto colocada na lista de seleção:

```
CREATE FUNCTION listar_filhos(text) RETURNS SETOF text AS $$
    SELECT nome FROM nodos WHERE pai = $1
$$ LANGUAGE SQL;
```

```
SELECT * FROM nodos;
```

```
nome    | pai
-----+-----
Topo    |
Filho1  | Topo
Filho2  | Topo
Filho3  | Topo
SubFilho1 | Filho1
SubFilho2 | Filho1
(6 linhas)
```

```
SELECT listar_filhos('Topo');
```

```
listar_filhos
-----
Filho1
Filho2
Filho3
(3 linhas)
```

```
SELECT nome, listar_filhos(nome) FROM nodos;
```



```

nome | listar_filhos
-----+-----
Top   | Filho1
Top   | Filho2
Top   | Filho3
Filho1 | SubFilho1
Filho1 | SubFilho2
(5 linhas)

```

Deve ser observado que no último comando `SELECT` não é mostrada nenhuma linha de saída para `Filho2`, `Filho3`, etc. Isto acontece porque `listar_filhos` retorna um conjunto vazio para estes argumentos e, portanto, não é gerada nenhuma linha de resultado.

### 31.4.5. Funções SQL polimórficas

As funções SQL podem ser declaradas como recebendo e retornando os tipos polimórficos `anyelement` e `anyarray`. Uma explicação mais detalhada sobre funções polimórficas pode ser vista na Seção 31.2.5. Abaixo está mostrada a função polimórfica `constroi_matriz`, que constrói uma matriz a partir de dois elementos com tipo de dado arbitrário:

```

CREATE FUNCTION constroi_matriz(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT constroi_matriz(1, 2) AS intarray, constroi_matriz('a'::text, 'b') AS textarray;

intarray | textarray
-----+-----
{1,2}    | {a,b}
(1 linha)

```

Deve ser observada a utilização da conversão de tipo `'a'::text` para especificar que o argumento é do tipo `text`. Isto é necessário quando o argumento é apenas um literal cadeia de caracteres, uma vez que de outra forma seria tratado como sendo do tipo `unknown`, e uma matriz de `unknown` não é um tipo válido. Sem a conversão de tipo aconteceria um erro como este:

```
ERRO: não foi possível determinar o tipo de anyarray/anyelement porque a entrada possui o tipo "unknown"
```

É permitido ter argumentos polimórficos com tipo retornado estabelecido, mas o contrário não é permitido. Por exemplo:

```

CREATE FUNCTION eh_maior(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT eh_maior(1, 2);

eh_maior
-----
f
(1 linha)

```

```

CREATE FUNCTION funcao_invalida() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;

```

```
ERRO: não foi possível determinar o tipo de dado do resultado
```

```
DETALHE: Uma função retornando "anyarray" ou "anyelement" deve ter pelo menos
um argumento do mesmo tipo.
```

## 31.5. Sobrecarga de função

Pode ser definida mais de uma função possuindo o mesmo nome SQL, desde que os argumentos recebidos sejam diferentes. Em outras palavras, os nomes das funções podem ser *sobrecarregados*. Quando um comando é executado, o

servidor determina a função a ser chamada a partir dos tipos de dado e do número de argumentos fornecidos. A sobrecarga também pode ser utilizada para simular funções com número variável de argumentos, até um número máximo finito.

Ao ser criada uma família de funções sobrecarregadas, deve ser tomado cuidado para não criar ambigüidades. Por exemplo, dadas as funções

```
CREATE FUNCTION teste(int, real) RETURNS ...
CREATE FUNCTION teste(smallint, double precision) RETURNS ...
```

não fica imediatamente claro qual das duas funções deve ser chamada por uma entrada trivial como `teste(1, 1.5)`. As regras de resolução implementadas atualmente estão descritas no Capítulo 10, mas não é prudente projetar um sistema que dependa de sutilezas deste comportamento.

A função que recebe um único argumento de tipo composto geralmente não deve ter o nome de nenhum atributo (campo) deste tipo. Lembre-se que `atributo(tabela)` é considerado equivalente a `tabela.atributo`. No caso de haver ambigüidade entre a função de tipo composto e um atributo de tipo composto, sempre será utilizado o atributo. É possível mudar esta escolha qualificando o nome da função com o esquema (ou seja, `esquema.func(tabela)`), mas é melhor evitar este problema não escolhendo nomes conflitantes.

Ao sobrecarregar funções na linguagem C, existe uma restrição adicional: o nome C de cada uma das funções da família de funções sobrecarregadas deve ser diferente dos nomes C de todas as outras funções, sejam internas ou carregadas dinamicamente. Se esta regra for violada, o comportamento não é portátil. Deve ser recebido um erro de ligação em tempo de execução, ou uma das funções será chamada (geralmente a interna). A forma alternativa da cláusula `AS` para o comando SQL `CREATE FUNCTION` desvincula o nome SQL da função do nome da função no código fonte C. Por exemplo,

```
CREATE FUNCTION teste(int) RETURNS int
    AS 'nome_do_arquivo', 'teste_larg'
    LANGUAGE C;
CREATE FUNCTION teste(int, int) RETURNS int
    AS 'nome_do_arquivo', 'teste_2arg'
    LANGUAGE C;
```

Os nomes das funções C neste exemplo refletem uma das várias convenções possíveis.

## 31.6. Categorias de volatilidade de função

Toda função é classificada quanto à sua *volatilidade*, sendo que as possibilidades são `VOLATILE` (volátil), `STABLE` (estável) e `IMMUTABLE` (imutável). Quando o comando `CREATE FUNCTION` não especifica a categoria, o padrão é `VOLATILE`. A categorização quanto à volatilidade é uma promessa feita ao otimizador sobre o comportamento da função:

- Uma função `VOLATILE` pode fazer qualquer coisa, inclusive modificar o banco de dados. Pode retornar resultados diferentes em chamadas sucessivas usando os mesmos argumentos. O otimizador não assume nada com relação ao comportamento destas funções. Um comando que utiliza uma função volátil reavalia a função em todas as linhas onde seu valor seja necessário.
- Uma função `STABLE` não pode modificar o banco de dados, e há garantia do retorno dos mesmos resultados quando a função recebe os mesmos parâmetros em todas as chamadas dentro de um mesmo comando envolvente. Esta categoria permite que o otimizador otimize o caso de várias chamadas à função dentro de um mesmo comando. Em particular, é seguro utilizar uma expressão contendo uma função deste tipo em uma condição de varredura de índice (Como a varredura de índice determina o valor a ser comparado somente uma vez, e não uma vez para cada linha, não é válido utilizar uma função `VOLATILE` em uma condição de varredura de índice).
- Uma função `IMMUTABLE` não pode modificar o banco de dados, e há garantia do retorno dos mesmos resultados quando esta recebe os mesmos argumentos. Esta categoria de função permite ao otimizador pré-avaliar a função quando o comando chama a função com argumentos constantes. Por exemplo, uma consulta do tipo `SELECT ... WHERE x = 2 + 2` pode ser transformada imediatamente em `SELECT ... WHERE x = 4`, porque a função subjacente ao operador de adição inteira está marcada como `IMMUTABLE`.

Para obter os melhores resultados de otimização, deve-se classificar a função na categoria de volatilidade que for mais rigorosa para a mesma.

Toda função com efeitos colaterais *deve* ser categorizada como `VOLATILE`, para que as chamadas às mesmas não sejam otimizadas. Mesmo as funções sem efeitos colaterais precisam ser classificadas como `VOLATILE`, se o valor retornado puder mudar dentro de um mesmo comando; alguns exemplos são `random()`, `currval()` e `timeofday()`.

Existe pouca diferença entre as categorias `STABLE` e `IMMUTABLE`, quando se considera comandos interativos simples que são planejados e executados imediatamente: não faz muita diferença se a função é executada durante o planejamento ou durante o início da execução do comando, mas existe muita diferença quando o plano é salvo para reutilização posterior. Classificar a função como `IMMUTABLE`, quando na verdade esta não é, pode fazer com que a mesma seja transformada prematuramente em uma constante durante o planejamento, resultando na utilização de um valor desatualizado durante as próximas utilizações do plano. Isto é um risco associado ao uso de declarações preparadas e ao uso de funções escritas em linguagens que colocam planos no `cache` (como o PL/pgSQL).

Devido ao comportamento de instantâneo do MVCC (consulte o Capítulo 12), uma função contendo apenas comandos `SELECT` pode ser classificada como `STABLE` com segurança, mesmo que faça seleção em tabelas que podem estar sendo modificadas por comandos simultâneos. O PostgreSQL executa a função `STABLE` utilizando o instantâneo estabelecido para o comando que faz a chamada e, portanto, será enxergada uma visão fixa do banco de dados durante o comando. Deve ser observado, também, que a família de funções `current_timestamp` é classificada como `estável`, uma vez que seus resultados não mudam dentro de uma transação.

O mesmo comportamento de instantâneo é utilizado nos comandos `SELECT` dentro das funções `IMMUTABLE`. Geralmente não se aconselha fazer seleções em tabelas do banco de dados dentro de uma função `IMMUTABLE`, uma vez que a imutabilidade será quebrada se o conteúdo da tabela mudar. Entretanto, o PostgreSQL não exige que seja assim.

Um erro comum é classificar a função como `IMMUTABLE` quando seu resultado depende de parâmetros de configuração. Por exemplo, uma função que manipula carimbos do tempo pode ter resultados dependentes da definição de `timezone`. Por motivo de segurança, estas funções devem ser classificadas como `STABLE`.

**Nota:** Antes da versão 8.0 do PostgreSQL, o requisito que as funções `STABLE` e `IMMUTABLE` não podem modificar o banco de dados não era requerido pelo sistema. A versão 8.0 obriga obedecer este requisito, requerendo que as funções SQL, e as funções escritas em linguagem procedural, destas categorias, não contenham nenhum comando SQL além do `SELECT` (Isto não é um teste totalmente seguro, uma vez que estas funções podem chamar funções `VOLATILE` que modificam o banco de dados. Se isto for feito, vai ser descoberto que as funções `STABLE` e `IMMUTABLE` não percebem as mudanças feitas no banco de dados aplicadas pela função chamada).

## 31.7. Funções nas linguagens procedurais

O PostgreSQL permite que funções definidas pelo usuário possam ser escritas em outras linguagens além de SQL e C. Estas outras linguagens são chamadas genericamente de *linguagens procedurais* (PLs). As linguagens procedurais não são construídas dentro do servidor PostgreSQL; são oferecidas como módulos carregáveis. Para obter informações adicionais deve ser consultado o Capítulo 34 e os seguintes.

## 31.8. Funções internas

As funções internas são funções escritas na linguagem C que foram ligadas estaticamente ao servidor PostgreSQL. O “corpo” da definição da função especifica o nome da função na linguagem C, que não precisa ser o mesmo nome declarado para uso no SQL (Por razões de compatibilidade com as versões anteriores, um corpo vazio é aceito como significando que o nome da função na linguagem C é o mesmo nome na linguagem SQL).

Normalmente, todas as funções internas presentes no servidor são declaradas durante a inicialização do agrupamento de bancos de dados (`initdb`), mas o usuário pode utilizar o comando `CREATE FUNCTION` para criar nomes aliases adicionais para uma função interna. As funções internas são declaradas em `CREATE FUNCTION` com o nome de linguagem `internal`. Por exemplo, para criar um aliás para a função `sqrt`:

```
CREATE FUNCTION raiz_quadrada(double precision) RETURNS double precision
AS 'dsqrt'
LANGUAGE internal
STRICT;

SELECT raiz_quadrada(9.61);
```

```

raiz_quadrada
-----
          3.1
(1 linha)

```

(A maioria das funções internas esperam ser declaradas como “strict”)

**Nota:** Nem todas as funções “pré-definidas” são “internas” no sentido acima. Algumas funções pré-definidas são escritas em SQL.

## 31.9. Funções na linguagem C

As funções definidas pelo usuário podem ser escritas em C (ou numa linguagem que possa ser tornada compatível com a linguagem C, como C++). Estas funções são compiladas em objetos carregáveis dinamicamente (também chamados de bibliotecas compartilhadas), sendo carregadas pelo servidor conforme haja necessidade. A funcionalidade de carregamento dinâmico é o que distingue as funções na “linguagem C” das funções “internas” — as convenções de codificação são essencialmente as mesmas para ambas (portanto, a biblioteca padrão de funções internas é uma preciosa fonte de exemplos de codificação para funções na linguagem C definidas pelo usuário).

Atualmente são utilizadas duas convenções de chamada diferentes para as funções em C. A convenção de chamada mais nova, “versão 1”, é indicada pela inclusão da chamada de macro `PG_FUNCTION_INFO_V1()` na função, conforme mostrado abaixo. A ausência desta macro indica uma função no estilo antigo (“versão 0”). Nestes dois casos o nome da linguagem especificado em `CREATE FUNCTION` é C. As funções no estilo antigo estão em obsolescência por causa de problemas de portabilidade e ausência de funcionalidades, mas ainda são aceitas por motivo de compatibilidade.

### 31.9.1. Carregamento dinâmico

Na primeira vez que uma função definida pelo usuário, presente em um arquivo objeto carregável, é chamada em uma sessão, o carregador dinâmico carrega o arquivo objeto na memória para que a função possa ser chamada. O comando `CREATE FUNCTION` para uma função C definida pelo usuário deve, portanto, especificar duas informações para a função: o nome do arquivo objeto carregável, e o nome C (símbolo de ligação), dentro do arquivo objeto, da função a ser chamada. Se o nome C não for especificado explicitamente, então é assumido como sendo o mesmo nome da função SQL.

É utilizado o seguinte algoritmo para localizar o arquivo objeto compartilhado baseado no nome fornecido no comando `CREATE FUNCTION`:

1. Se o nome for um caminho absoluto, o arquivo especificado é carregado.
2. Se o nome começar pela cadeia de caracteres `$libdir`, esta parte é substituída pelo diretório de biblioteca do pacote PostgreSQL, determinado em tempo de construção.
3. Se o nome não contiver a parte do diretório, o arquivo é procurado no caminho especificado pela variável de configuração `dynamic_library_path`.
4. Senão (o arquivo não foi encontrado no caminho, ou contém a parte de diretório não-absoluta), o carregador dinâmico tenta usar o nome conforme especificado, o que quase certamente não vai ser bem-sucedido (Não é confiável depender do diretório de trabalho corrente).

Se esta sequência não for bem-sucedida, a extensão de nome de arquivo de biblioteca compartilhada específica da plataforma (geralmente `.so`) é anexada ao nome fornecido, e esta sequência é tentada novamente. Se também não for bem-sucedida, então o carregamento falha.

O ID do usuário sob o qual o PostgreSQL executa deve ser capaz de percorrer o caminho até o arquivo que se deseja carregar. Tornar o arquivo ou um diretório de nível mais alto não legível e/ou não executável pelo usuário postgres é um erro comum.

De qualquer forma, o nome do arquivo fornecido no comando `CREATE FUNCTION` é gravado literalmente nos catálogos do sistema e, portanto, se for necessário carregar o arquivo novamente o mesmo procedimento é aplicado.

**Nota:** O PostgreSQL não compila uma função C automaticamente. O arquivo objeto deve ser compilado antes de ser referenciado no comando `CREATE FUNCTION`. Para obter informações adicionais deve ser consultada a Seção 31.9.6.

O arquivo objeto carregável dinamicamente é mantido na memória após ter sido utilizado pela primeira vez. As chamadas posteriores às funções presentes neste arquivo, na mesma sessão, somente causam um pequeno trabalho extra de procura na

tabela de símbolos. Se for necessário obrigar uma nova carga do arquivo objeto, por exemplo após este ser recompilado, deve ser utilizado o comando `LOAD`, ou iniciada uma nova sessão.

Recomenda-se que as bibliotecas compartilhadas tenham posição relativa a `$libdir`, ou estejam no caminho de biblioteca dinâmica, simplificando atualizações de versão se a nova instalação estiver em um local diferente. O diretório real representado por `$libdir` pode ser descoberto através do comando `pg_config --pkglibdir`.<sup>2</sup>

Antes da versão 7.2 do PostgreSQL, somente era possível especificar no comando `CREATE FUNCTION` caminhos absolutos exatos para os arquivos objeto. Esta modalidade está em obsolescência, uma vez que torna a definição da função não portátil sem necessidade. É melhor especificar apenas o nome da biblioteca compartilhada sem caminho nem extensão, e deixar o mecanismo de procura fornecer estas informações.

### 31.9.2. Tipos base em funções na linguagem C

Para saber como escrever funções na linguagem C é necessário saber como o PostgreSQL representa internamente os tipos de dado base, e como estes podem ser passados de/para as funções. Internamente, o PostgreSQL considera o tipo base como um “objeto binário grande de memória” (blob of memory). Por sua vez, as funções definidas pelo usuário para o tipo definem a maneira como o PostgreSQL pode operar o tipo, ou seja, o PostgreSQL somente armazena e busca os dados do disco, e usa as funções definidas pelo usuário para entrada, processamento e saída dos dados.

Os tipos base podem ter um destes três formatos internos:

- passado por valor, comprimento fixo
- passado por referência, comprimento fixo
- passado por referência, comprimento variável

Os tipos passados por valor podem ter comprimento de 1, 2 ou 4 bytes apenas (também de 8 bytes, se na máquina `sizeof(Datum)` for 8). Deve-se tomar cuidado para que os tipos definidos pelo usuário sejam de tal forma que possuam o mesmo tamanho (em bytes) em todas as arquiteturas. Por exemplo, o tipo `long` é perigoso, porque possui 4 bytes em algumas máquinas e 8 bytes em outras, enquanto o tipo `int` possui 4 bytes na maioria das máquinas Unix. Uma implementação razoável do tipo `int4` em uma máquina Unix pode ser:

```
/* inteiro de 4 bytes, passado por valor */
typedef int int4;
```

Por outro lado, os tipos de comprimento fixo, de qualquer tamanho, podem ser passados por referência. Por exemplo, abaixo está mostrada a implementação de um tipo do PostgreSQL:

```
/* estrutura de 16 bytes, passada por referência */
typedef struct
{
    double x, y;
} Point;
```

Somente podem ser utilizados ponteiros para estes tipos para passá-los de/para as funções do PostgreSQL. Para retornar o valor de um tipo como este, é alocada a quantidade correta de memória com `palloc`, preenchida a memória alocada, e retornado um ponteiro para a memória alocada (Também pode ser retornado diretamente um valor de entrada, que possua o mesmo tipo do valor retornado, retornando um ponteiro para o valor de entrada. Entretanto, não se deve modificar *nunca* o conteúdo de um valor passado por referência).

Por fim, todos os tipos de comprimento variável também devem ser passados por referência. Todos os tipos de comprimento variável devem começar por um campo de comprimento, contendo exatamente 4 bytes, e todos os dados a serem armazenados dentro deste tipo devem estar localizados na memória logo após o campo de comprimento. O campo de comprimento contém o comprimento total da estrutura, ou seja, inclui também o tamanho do próprio campo de comprimento.

Como exemplo, o tipo `text` pode ser definido da seguinte forma:

```
typedef struct {
    int4 comprimento;
    char dado[1];
} text;
```

Obviamente, o campo dado declarado não possui comprimento suficiente para armazenar todas as cadeias de caracteres possíveis. Como não é possível declarar estruturas de tamanho variável em C, dependemos do conhecimento de que o compilador C não verifica o intervalo dos índices das matrizes. Simplesmente é alocada a quantidade necessária de espaço, e depois a matriz é acessada como se tivesse sido declarada com o comprimento correto (Este é um truque comum, que pode ser visto em muitos livros texto sobre C).

Ao manipular tipos de comprimento variável, deve-se tomar o cuidado de alocar a quantidade correta de memória, e definir o campo de comprimento corretamente. Por exemplo, se for desejado armazenar 40 bytes em uma estrutura `text`, pode ser utilizado um fragmento de código como este:

```
#include "postgres.h"
...
char buffer[40]; /* nosso dado de origem */
...
text *destino = (text *) palloc(VARHDRSZ + 40);
destino->comprimento = VARHDRSZ + 40;
memcpy(destino->dado, buffer, 40);
...
```

`VARHDRSZ` é o mesmo que `sizeof(int4)`, mas é considerado um bom estilo utilizar a macro `VARHDRSZ` para fazer referência ao tamanho adicional para o tipo de comprimento variável.

A Tabela 31-1 especifica qual tipo C corresponde a qual tipo SQL, quando se escreve uma função na linguagem C que utiliza um tipo interno do PostgreSQL. A coluna “Definido em” informa o arquivo de cabeçalho que deve ser incluído para obter a definição do tipo (Na verdade, a definição pode estar em um outro arquivo incluído pelo arquivo informado. Recomenda-se aos usuários que se limitem à interface definida). Deve ser observado que sempre deve ser incluído primeiro, em todos os arquivos fonte, `postgres.h`, porque este declara várias outras coisas que são sempre necessárias de alguma forma.

**Tabela 31-1. Tipos C equivalentes aos tipos SQL internos**

Tipo SQL	Tipo C	Definido em
abstime	AbsoluteTime	utils/nabstime.h
boolean	bool	postgres.h (talvez interno do compilador)
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	(interno do compilador)
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
smallint (int2)	int2 ou int16	postgres.h
int2vector	int2vector*	postgres.h
integer (int4)	int4 ou int32	postgres.h
real (float4)	float4*	postgres.h
double precision (float8)	float8*	postgres.h
interval	Interval*	utils/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
oid	Oid	postgres.h

Tipo SQL	Tipo C	Definido em
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	regproc	postgres.h
reltime	RelativeTime	utils/nabstime.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp*	utils/timestamp.h
tinterval	TimeInterval	utils/nabstime.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

Agora que já foram examinadas todas as estruturas possíveis para os tipos base, podem ser mostrados alguns exemplos de funções verdadeiras.

### 31.9.3. Convenções de chamada Versão-0 para funções na linguagem C

Será apresentado primeiro o “estilo antigo” de convenção de chamada — embora esta modalidade esteja em obsolescência, é mais fácil começar por ela. No método versão-0 os argumentos e o resultado da função C são simplesmente declarados no estilo C usual, mas tomando cuidado para utilizar a representação C de cada tipo de dado SQL conforme mostrado acima.

Abaixo estão mostrados alguns exemplos:

```
#include "postgres.h"
#include <string.h>

/* Por valor */

int
somar_um(int arg)
{
    return arg + 1;
}

/* Por referência, comprimento fixo */

float8 *
somar_um_float8(float8 *arg)
{
    float8    *resultado = (float8 *) palloc(sizeof(float8));

    *resultado = *arg + 1.0;

    return resultado;
}

Point *
construir_ponto(Point *pontox, Point *pontoy)
{

```

```

Point      *novo_ponto = (Point *) malloc(sizeof(Point));

novo_ponto->x = pontox->x;
novo_ponto->y = pontoy->y;

return novo_ponto;
}

/* Por referência, comprimento variável */

text *
copiar_texto(text *t)
{
    /*
     * VARSIZE é o tamanho total da estrutura em bytes.
     */
    text *novo_t = (text *) malloc(VARSIZE(t));
    VARATT_SIZEP(novo_t) = VARSIZE(t);
    /*
     * VARDATA é o ponteiro para a região de dados da estrutura.
     */
    memcpy((void *) VARDATA(novo_t), /* destino */
           (void *) VARDATA(t),      /* origem */
           VARSIZE(t)-VARHDRSZ);     /* quantidade de bytes */
    return novo_t;
}

text *
concatenar_texto(text *arg1, text *arg2)
{
    int32 tamanho_novo_texto = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *novo_texto = (text *) malloc(tamanho_novo_texto);

    VARATT_SIZEP(novo_texto) = tamanho_novo_texto;
    memcpy(VARDATA(novo_texto), VARDATA(arg1), VARSIZE(arg1)-VARHDRSZ);
    memcpy(VARDATA(novo_texto) + (VARSIZE(arg1)-VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2)-VARHDRSZ);
    return novo_texto;
}

```

Supondo que o código acima tenha sido escrito no arquivo `funcs.c` e compilado dentro de um objeto compartilhado, as funções poderiam ser definidas no PostgreSQL usando comandos como estes:

```

CREATE FUNCTION somar_um(integer) RETURNS integer
AS 'DIRETÓRIO/funcs', 'somar_um'
LANGUAGE C STRICT;

-- observe a sobrecarga do nome de função SQL "somar_um"
CREATE FUNCTION somar_um(double precision) RETURNS double precision
AS 'DIRETÓRIO/funcs', 'somar_um_float8'
LANGUAGE C STRICT;

CREATE FUNCTION construir_ponto(point, point) RETURNS point
AS 'DIRETÓRIO/funcs', 'construir_ponto'
LANGUAGE C STRICT;

CREATE FUNCTION copiar_texto(text) RETURNS text
AS 'DIRETÓRIO/funcs', 'copiar_texto'
LANGUAGE C STRICT;

CREATE FUNCTION concatenar_texto(text, text) RETURNS text
AS 'DIRETÓRIO/funcs', 'concatenar_texto',

```



```
LANGUAGE C STRICT;
```

Neste caso, *DIRETÓRIO* representa o diretório do arquivo de biblioteca compartilhada (por exemplo, o diretório do tutorial do PostgreSQL que contém o código dos exemplos utilizados nesta seção) (Um estilo melhor seria utilizar apenas 'funcs' na cláusula AS, após adicionar *DIRETÓRIO* ao caminho de procura. Em todo caso, pode ser omitida a extensão específica do sistema para a biblioteca compartilhada, normalmente .so ou .sl).

Deve ser observado que as funções foram especificadas como “strict”, significando que o sistema deve assumir, automaticamente, um resultado nulo se algum valor de entrada for nulo. Fazendo assim, evita-se a necessidade de verificar entradas nulas no código da função. Sem isto, seria necessário verificar os valores nulos explicitamente como, por exemplo, verificando a existência de um ponteiro nulo para cada argumento passado por referência (Para os argumentos passados por valor, não haveria nem como verificar!).

Embora esta convenção de chamada seja simples de usar, não é muito portátil; em algumas arquiteturas ocorrem problemas ao se passar tipos de dado menores que int desta forma. Também, não existe nenhuma forma simples de retornar um resultado nulo, nem para lidar com argumentos nulos de alguma outra forma que não seja tornando a função estrita. A convenção versão-1, apresentada a seguir, supera estas limitações.

### 31.9.4. Convenções de chamada Versão-1 para as funções na linguagem C

A convenção de chamada versão-1 se baseia em macros que suprimem a maior parte da complexidade da passagem de argumentos e resultados. A declaração C de uma função versão-1 é sempre

```
Datum nome_da_função(PG_FUNCTION_ARGS)
```

Além disso, a chamada de macro

```
PG_FUNCTION_INFO_V1(nome_da_função);
```

deve aparecer no mesmo arquivo fonte (por convenção é escrita logo antes da própria função). Esta chamada de macro não é necessária para as funções na linguagem internal, uma vez que o PostgreSQL assume que todas as funções internas usam a convenção versão-1. Entretanto, é requerido nas funções carregadas dinamicamente.

Em uma função versão-1, cada argumento é buscado utilizando a macro PG\_GETARG\_xxx() correspondente ao tipo de dado do argumento, e o resultado é retornado utilizando a macro PG\_RETURN\_xxx() para o tipo retornado. A macro PG\_GETARG\_xxx() recebe como argumento o número do argumento da função a ser buscado, contado a partir de 0. A macro PG\_RETURN\_xxx() recebe como argumento o valor a ser retornado.

Abaixo estão mostradas as mesmas funções vistas acima, codificadas no estilo versão-1:

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"

/* Por valor */

PG_FUNCTION_INFO_V1(somar_um);

Datum
somar_um(PG_FUNCTION_ARGS)
{
    int32    arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* Por referência, comprimento fixo */

PG_FUNCTION_INFO_V1(somar_um_float8);

Datum
somar_um_float8(PG_FUNCTION_ARGS)
{
```

```

/* As macros para FLOAT8 escondem sua natureza de passado por referência. */
float8   arg = PG_GETARG_FLOAT8(0);

PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(construir_ponto);

Datum
construir_ponto(PG_FUNCTION_ARGS)
{
    /* Aqui a natureza de passado por referência de Point não é escondida. */
    Point   *pontox = PG_GETARG_POINT_P(0);
    Point   *pontoy = PG_GETARG_POINT_P(1);
    Point   *novo_ponto = (Point *) palloc(sizeof(Point));

    novo_ponto->x = pontox->x;
    novo_ponto->y = pontoy->y;

    PG_RETURN_POINT_P(novo_ponto);
}

/* Por referência, comprimento variável */

PG_FUNCTION_INFO_V1(copiar_texto);

Datum
copiar_texto(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_P(0);
    /*
     * VARSIZE é o tamanho total da estrutura em bytes.
     */
    text     *novo_t = (text *) palloc(VARSIZE(t));
    VARATT_SIZEP(novo_t) = VARSIZE(t);
    /*
     * VARDATA é o ponteiro para a região de dados da estrutura.
     */
    memcpy((void *) VARDATA(novo_t), /* destino */
           (void *) VARDATA(t),      /* origem */
           VARSIZE(t)-VARHDRSZ);     /* quantidade de bytes */
    PG_RETURN_TEXT_P(novo_t);
}

PG_FUNCTION_INFO_V1(concatenar_texto);

Datum
concatenar_texto(PG_FUNCTION_ARGS)
{
    text *arg1 = PG_GETARG_TEXT_P(0);
    text *arg2 = PG_GETARG_TEXT_P(1);
    int32 tamanho_novo_texto = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *novo_texto = (text *) palloc(tamanho_novo_texto);

    VARATT_SIZEP(novo_texto) = tamanho_novo_texto;
    memcpy(VARDATA(novo_texto), VARDATA(arg1), VARSIZE(arg1)-VARHDRSZ);
    memcpy(VARDATA(novo_texto) + (VARSIZE(arg1)-VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2)-VARHDRSZ);
    PG_RETURN_TEXT_P(novo_texto);
}

```

Os comandos `CREATE FUNCTION` são idênticos aos das funções versão-0 equivalentes.

À primeira vista, as convenções de codificação versão-1 podem parecer apenas um obscurantismo sem sentido. Entretanto, oferecem várias melhorias porque as macros podem esconder detalhes desnecessários. Como exemplo podemos citar a codificação de `somar_um_float8`, onde não é mais necessário se preocupar com o fato de `float8` ser um tipo passado por referência. Outro exemplo é que as macros `GETARG` para tipos de comprimento variável permitem buscar valores “toasted” (comprimidos ou fora de linha) de forma mais eficiente.

Uma grande melhoria nas funções versão-1 é o tratamento melhor das entradas e resultados nulos. A macro `PG_ARGISNULL(n)` permite à função testar se cada um dos valores de entrada é nulo (obviamente, só é necessário nas funções não declaradas como “strict”). Nas macros `PG_GETARG_xxx()` os argumentos de entrada são contados a partir de zero. Deve ser observado que deve ser evitado executar `PG_GETARG_xxx()` até que tenha sido constatado que o argumento não é nulo. Para retornar um resultado nulo deve ser executado `PG_RETURN_NULL()`; isto funciona tanto nas funções estritas quanto nas não estritas.

Outras opções fornecidas para a interface no novo estilo são duas variantes das macros `PG_GETARG_xxx()`. A primeira delas, `PG_GETARG_xxx_COPY()` garante retornar uma cópia do argumento especificado onde é possível escrever com segurança (As macros comuns, algumas vezes, retornam um ponteiro para um valor que está fisicamente armazenado em uma tabela e, portanto, não é possível escrever neste local. A utilização das macros `PG_GETARG_xxx_COPY()` garante um resultado onde pode ser escrito).

A segunda variante consiste nas macros `PG_GETARG_xxx_SLICE()` que recebem três parâmetros. O primeiro é o número do argumento da função (como acima). O segundo e o terceiro são o deslocamento e o comprimento do segmento a ser retornado. Os deslocamentos são contados a partir de zero, e um comprimento negativo requer que o restante do valor seja retornado. Estas macros proporcionam um acesso mais eficiente às partes de valores grandes, caso estes possuam um tipo de armazenamento “external” (O tipo de armazenamento de uma coluna pode ser especificado utilizando `ALTER TABLE nome_da_tabela ALTER COLUMN nome_da_coluna SET STORAGE tipo_de_armazenamento`. O `tipo_de_armazenamento` é um entre `plain`, `external`, `extended` ou `main`).

Por fim, as convenções de chamada de função versão-1 tornam possível retornar “conjuntos” (`set`) como resultados (Seção 31.9.10), implementar funções de gatilho (Capítulo 32) e tratadores de chamada de linguagem procedural (Capítulo 45). Também, o código versão-1 é mais portátil que o código versão-0, porque não quebra as restrições do protocolo de chamada de função padrão do padrão C. Para obter informações adicionais deve ser consultado o arquivo `src/backend/utils/fmgr/README` na distribuição do código fonte.

### 31.9.5. Escrita de código

Antes de passar para os tópicos mais avançados, devem ser vistas algumas regras de codificação para funções escritas na linguagem C no PostgreSQL. Embora seja possível carregar funções escritas em outras linguagens diferentes da linguagem C no PostgreSQL, geralmente é difícil (quando não é totalmente impossível) porque as outras linguagens, como C++, FORTRAN e Pascal, geralmente não seguem a mesma convenção de chamada da linguagem C. Ou seja, as outras linguagens não passam argumentos e retornam os valores das funções da mesma maneira. Por este motivo, será assumido que as funções escritas na linguagem C são realmente escritas em C.

As regras básicas para construir funções na linguagem C são as seguintes:

- Utilizar `pg_config --includedir-server` para descobrir onde os arquivos de cabeçalho do servidor PostgreSQL estão instalados no sistema (ou no sistema utilizado pelos seus usuários). <sup>3</sup> Esta opção passou a existir a partir do PostgreSQL 7.2. Para o PostgreSQL 7.1 deve ser utilizada a opção `--includedir` (o comando `pg_config` termina com status diferente de zero quando encontra uma opção desconhecida). Para as versões anteriores a 7.1 é necessário descobrir por si próprio, mas como estas versões são anteriores à introdução das convenções de chamada corrente, não é provável que se deseje dar suporte a estas versões.
- Para alocar memória devem ser utilizadas as funções `palloc` e `pfree` do PostgreSQL, em vez das funções correspondentes da biblioteca C `malloc` e `free`. A memória alocada por `palloc` é liberada automaticamente ao término de cada transação, evitando perda de memória.
- Os bytes das estruturas devem ser sempre zerados utilizando `memset`. Se isto não for feito, ficará difícil suportar índices `hash` ou junções `hash`, uma vez que devem ser pegos somente os bits significativos da estrutura de dados para calcular o `hash`. Mesmo se forem inicializados todos os campos da estrutura, poderá haver preenchimento de alinhamento (buracos na estrutura) contendo sujeira.

- A maioria dos tipos internos do PostgreSQL estão declarados em `postgres.h`, enquanto as interfaces de gerência de função (`PG_FUNCTION_ARGS`, etc.) estão declaradas em `fmgr.h`, portanto é necessário incluir ao menos estes dois arquivos. Por motivo de portabilidade é melhor incluir *primeiro* `postgres.h`, antes de qualquer outro arquivo de cabeçalho do sistema ou do usuário. Ao se incluir `postgres.h` também são incluídos `elog.h` e `palloc.h`.
- Os nomes dos símbolos definidos dentro dos arquivos objeto não devem conflitar entre si ou com os símbolos definidos no executável do servidor PostgreSQL. As funções e variáveis deverão ser renomeadas se aparecerem mensagens de erro neste sentido.
- Compilar e ligar o código objeto para que possa ser carregado dinamicamente no PostgreSQL sempre requer sinalizadores especiais. Consulte a Seção 31.9.6 para obter uma explicação detalhada sobre como isto é feito no seu sistema operacional em particular.

### 31.9.6. Compilar e ligar as funções carregadas dinamicamente

Antes que as funções escritas em C para estender o PostgreSQL possam ser utilizadas, estas funções precisam ser compiladas e ligadas de uma forma especial para produzir um arquivo que possa ser carregado dinamicamente pelo servidor. Para ser exato, precisa ser criada uma *biblioteca compartilhada*.

Para obter informações além das contidas nesta seção deve ser lida a documentação do sistema operacional, em particular as páginas do manual do compilador C, `gcc`, e do editor de ligação, `ld`. Além disso, o código fonte do PostgreSQL contém vários exemplos funcionais no diretório `contrib`. Entretanto, a dependência destes exemplos torna os módulos dependentes da disponibilidade do código fonte do PostgreSQL.

Criar bibliotecas compartilhadas geralmente é análogo a ligar executáveis: primeiro os arquivos fonte são compilados em arquivos objetos e, depois, os arquivos objeto são ligados. Os arquivos objeto precisam ser criados como *código independente de posição* (PIC), o que significa conceitualmente que podem ser colocados em uma posição de memória arbitrária ao serem carregados pelo executável (Os arquivos objeto destinados a executáveis geralmente não são compilados deste modo). O comando para ligar uma biblioteca compartilhada contém sinalizadores especiais para distinguir da ligação de um executável (ao menos em teoria — em alguns sistemas a prática é muito mais feia).

Nos exemplos a seguir é assumido que o código fonte está no arquivo `foo.c`, e que é criada a biblioteca compartilhada `foo.so`. O arquivo objeto intermediário se chama `foo.o`, a não ser que seja dito o contrário. A biblioteca compartilhada pode conter mais de um arquivo objeto, mas aqui é utilizado apenas um arquivo.

#### BSD/OS

O sinalizador do compilador para criar PIC é `-fpic`. O sinalizador do ligador para criar biblioteca compartilhada é `-shared`.

```
gcc -fpic -c foo.c
ld -shared -o foo.so foo.o
```

Isto se aplica a partir da versão 4.0 do BSD/OS.

#### FreeBSD

O sinalizador do compilador para criar PIC é `-fpic`. Para criar bibliotecas compartilhadas o sinalizador do compilador é `-shared`.

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

Isto se aplica a partir a versão 3.0 do FreeBSD.

#### HP-UX

O sinalizador do compilador do sistema para criar PIC é `+z`. Quando se utiliza o GCC é `-fpic`. O sinalizador do ligador para criar bibliotecas compartilhadas é `-b`. Portanto

```
cc +z -c foo.c

ou

gcc -fpic -c foo.c

e depois

ld -b -o foo.sl foo.o
```

O HP-UX utiliza a extensão `.sl` para bibliotecas compartilhadas, diferentemente da maioria dos outros sistemas.

## IRIX

PIC é o padrão, não é necessário nenhum sinalizador especial do compilador. A opção do ligador para criar bibliotecas compartilhadas é `-shared`.

```
cc -c foo.c
ld -shared -o foo.so foo.o
```

## Linux

O sinalizador do compilador para criar PIC é `-fpic`. Em certas situações em algumas plataformas deve ser utilizado `-fPIC` se `-fpic` não funcionar. Deve ser consultado o manual do GCC para obter mais informações. O sinalizador do compilador para criar a biblioteca compartilhada é `-shared`. Um exemplo completo se parece com:

```
cc -fpic -c foo.c
cc -shared -o foo.so foo.o
```

## MacOS X

Abaixo segue um exemplo. É assumido que as ferramentas de desenvolvimento estão instaladas.

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

## NetBSD

O sinalizador do compilador para criar PIC é `-fpic`. Nos sistemas ELF, é utilizado o compilador com o sinalizador `-shared` para ligar as bibliotecas compartilhadas. Nos sistemas antigos, não-ELF, é utilizado `ld -Bshareable`.

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

## OpenBSD

O sinalizador do compilador para criar PIC é `-fpic`. É utilizado `ld -Bshareable` para ligar bibliotecas compartilhadas.

```
gcc -fpic -c foo.c
ld -Bshareable -o foo.so foo.o
```

## Solaris

O sinalizador do compilador para criar PIC é `-KPIC` quando é utilizado o compilador da Sun, e `-fpic` quando é utilizado o GCC. Para ligar bibliotecas compartilhadas a opção do compilador é `-G` para os dois compiladores ou, como alternativa, `-shared` quando é utilizado o GCC.

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o

ou

gcc -fpic -c foo.c
gcc -G -o foo.so foo.o
```

## Tru64 UNIX

PIC é o padrão e, portanto, o comando para compilar é o usual. É utilizado `ld` com opções especiais para ligar:

```
cc -c foo.c
ld -shared -expect_unresolved '*' -o foo.so foo.o
```

O mesmo procedimento é empregado quando GCC é utilizado no lugar do compilador do sistema; nenhuma opção especial é necessária.

## UnixWare

O sinalizador do compilador para criar PIC é `-K PIC` quando é utilizado o compilador da SCO, e `-fpic` no GCC. Para ligar bibliotecas compartilhadas a opção do compilador é `-G` quando é utilizado o compilador da SCO e `-shared` quando é utilizado o GCC.

```
cc -K PIC -c foo.c
cc -G -o foo.so foo.o
```

```
or
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

**Dica:** Caso julgue muito complicado, poderá ser levado em consideração a utilização da GNU Libtool (<http://www.gnu.org/software/libtool/>), que esconde as diferenças entre as plataformas atrás de uma interface uniforme.

O arquivo de biblioteca compartilhada produzido pode então ser carregado no PostgreSQL. Ao se especificar o nome do arquivo no comando `CREATE FUNCTION`, deve ser especificado o nome do arquivo da biblioteca compartilhada, e não do arquivo objeto intermediário. Deve ser observado que a extensão padrão do sistema para biblioteca compartilhada (geralmente `.so` ou `.sl`) pode ser omitida no comando `CREATE FUNCTION`, e normalmente deve ser omitida para uma melhor portabilidade.

Veja na Seção 31.9.1 onde o servidor espera encontrar os arquivos de biblioteca compartilhada.

### 31.9.7. Infraestrutura de construção de extensão

Caso se esteja pensando em distribuir os módulos de extensão do PostgreSQL, a implantação de um sistema de construção portátil para estes módulos pode ser bem difícil. Por isso, a instalação do PostgreSQL disponibiliza uma infraestrutura de construção de extensões, chamada PGXS, para que módulos de extensão simples possam ser construídos em um servidor já instalado. Deve ser observado que esta infraestrutura não tem por objetivo ser uma estrutura de construção de sistemas universal que possa ser utilizada para construir todos os sistemas que possuem interface com o PostgreSQL; esta infraestrutura simplesmente automatiza as regras comuns de construção para módulos de extensão do servidor simples. Para desenvolver pacotes mais complicados, é necessário escrever seu próprio sistema de construção.

Para utilizar esta infraestrutura para as próprias extensões, deve ser escrito um arquivo de construção, onde é necessário definir algumas variáveis e por fim incluir o arquivo de construção PGXS global. Abaixo está mostrado um exemplo que constrói um módulo de extensão chamado `isbn_issn` que consiste em uma biblioteca compartilhada, um script SQL, e um arquivo texto de documentação:

```
MODULES = isbn_issn
DATA_built = isbn_issn.sql
DOCS = README.isbn_issn

PGXS := $(shell pg_config --pgxs)
include $(PGXS)
```

As duas últimas linhas devem ser sempre as mesmas. Antes disso no arquivo podem ser definidas variáveis e adicionadas regras personalizadas para o `make`.

Podem ser definidas as seguintes variáveis:

```
MODULES
    lista dos objetos compartilhados a serem construídos a partir do arquivo fonte com o mesmo tronco (não devem ser
    incluídos sufixos nesta lista)

DATA
    arquivos aleatórios a serem instalados em prefix/share/contrib

DATA_built
    arquivos aleatórios a serem instalados em prefix/share/contrib, que primeiro devem ser construídos

DOCS
    arquivos aleatórios a serem instalados em prefix/doc/contrib

SCRIPTS
    arquivos de script (não binários) a serem instalados em prefix/bin

SCRIPTS_built
    arquivos de script (não binários) a serem instalados em prefix/bin, que primeiro devem ser construídos
```

REGRESS

lista de casos de teste de regressão (sem sufixo)

ou no máximo uma destas duas:

PROGRAM

o programa binário a ser construído (lista os arquivos objeto em OBJS)

MODULE\_big

o objeto compartilhado a ser construído (lista os arquivos objeto em OBJS)

Também podem ser definidas:

EXTRA\_CLEAN

arquivos adicionais a serem removidos por `make clean`

PG\_CPPFLAGS

adicionado a CPPFLAGS

PG\_LIBS

adicionado a linha de ligação de PROGRAM

SHLIB\_LINK

adicionado a linha de ligação de MODULE\_big

Este arquivo de construção deve ser criado com o nome de Makefile no diretório que contém a extensão. Depois pode ser executado `make` para compilar e, posteriormente, `make install` para instalar o módulo. A extensão é compilada e instalada na instalação do PostgreSQL que corresponde ao primeiro comando `pg_config` encontrado no caminho de procura.

### 31.9.8. Argumentos de tipo composto em funções na linguagem C

Os tipos compostos não possuem uma disposição fixa como as estruturas C. As instâncias de um tipo composto podem conter campos nulos. Além disso, os tipos compostos que são parte de uma hierarquia de herança podem possuir campos diferentes dos outros membros da mesma hierarquia de herança. Por isso, o PostgreSQL disponibiliza uma função de interface para acessar os campos dos tipos compostos a partir da linguagem C.

Suponha que desejamos escrever uma função para responder a consulta

```
SELECT nome, c_sobrepago(emp, 1500) AS sobrepago
FROM emp
WHERE name = 'José' OR name = 'João';
```

Utilizando as convenções de chamada versão 0, `c_sobrepago` poderia ser definida como:

```
#include "postgres.h"
#include "executor/executor.h" /* para GetAttributeByName() */

bool
c_sobrepago(HeapTupleHeader t, /* a linha corrente de emp */
            int32 limite)
{
    bool isnull;
    int32 salario;

    salario = DatumGetInt32(GetAttributeByName(t, "salario", &isnull));
    if (isnull)
        return false;
    return salario > limite;
}
```

Na codificação versão-1, o código acima ficaria parecido com:

```
#include "postgres.h"
#include "executor/executor.h" /* para GetAttributeByName() */

PG_FUNCTION_INFO_V1(c_sobrepago);

Datum
c_sobrepago(PG_FUNCTION_ARGS)
{
    HeapTupleHeader t = PG_GETARG_HEAPTUPLEHEADER(0);
    int32           limite = PG_GETARG_INT32(1);
    bool isnull;
    Datum salario;

    salario = GetAttributeByName(t, "salario", &isnull);
    if (isnull)
        PG_RETURN_BOOL(false);
    /* Como alternativa poderia se preferir usar PG_RETURN_NULL() para salário nulo. */

    PG_RETURN_BOOL(DatumGetInt32(salario) > limite);
}
```

`GetAttributeByName` é a função de sistema do PostgreSQL que retorna atributos da linha especificada. Possui três argumentos: o argumento do tipo `HeapTupleHeader` passado para a função, o nome do atributo desejado e o parâmetro retornado que informa se o atributo é nulo. `GetAttributeByName` retorna um valor do tipo `Datum` que pode ser convertido no tipo de dado apropriado utilizando a macro `DatumGetXXX()` apropriada. Deve ser observado que o valor retornado não tem sentido se o sinalizador de nulo estiver definido; sempre deve ser verificado o sinalizador de nulo antes de fazer qualquer coisa com o resultado.

Existe, também, a função `GetAttributeByNum` que seleciona o atributo de destino pelo número da coluna em vez de pelo nome.

O seguinte comando declara a função `c_sobrepago` no SQL:

```
CREATE FUNCTION c_sobrepago(emp, integer) RETURNS boolean
AS 'DIRETÓRIO/funcs', 'c_sobrepago'
LANGUAGE C STRICT;
```

Deve ser observado que foi utilizado `STRICT` e, portanto, não é necessário verificar se os argumentos são nulos.

### 31.9.9. Retorno de linhas (tipos compostos) por funções na linguagem C

Para retornar uma linha ou valor de tipo composto por uma função escrita na linguagem C, pode ser utilizada uma API especial que disponibiliza macros e funções que escondem a maior parte da complexidade envolvida na construção de tipos de dado compostos. Para utilizar esta API, deve ser incluído no código fonte:

```
#include "funcapi.h"
```

Existem duas maneiras de construir valor de dado composto (de agora em diante “tupla”): pode-se construir a partir de uma matriz de valores `Datum`, ou a partir de uma matriz de cadeias de caracteres `C` que possam ser passadas para as funções de conversão de entrada dos tipos de dado das colunas da tupla. Nestes dois casos, primeiro é necessário obter ou construir um descritor `TupleDesc` para a estrutura da tupla. Quando se trabalha com `Datum`, é passado o descritor `TupleDesc` para `BlessTupleDesc` e depois chamada `heap_formtuple` para cada linha. Quando se trabalha com cadeias de caracteres `C`, o descritor `TupleDesc` é passado para `TupleDescGetAttInMetadata` e depois chamada `BuildTupleFromCStrings` para cada linha. No caso de uma função que retorna um conjunto de tuplas, os passos de configuração podem ser todos feitos uma vez durante a primeira chamada à função.

Estão disponíveis várias funções de ajuda para configurar o descritor `TupleDesc` inicial. Se for desejado utilizar um tipo composto com nome, as informações podem ser buscadas nos catálogos do sistema. Deve ser utilizada

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

para obter `TupleDesc` para uma relação pelo nome, ou



```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

para obter `TupleDesc` baseado no OID do tipo. Pode ser utilizado para obter `TupleDesc` para um tipo base ou composto. Ao se escrever uma função que retorna `record`, o descritor `TupleDesc` esperado deve ser passado por quem faz a chamada.

Uma vez que se tenha construído `TupleDesc` deve ser chamada

```
TupleDesc BlessTupleDesc(TupleDesc tupdesc)
```

quando se planeja trabalhar com `Datum`, ou

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

quando se planeja trabalhar com cadeias de caracteres C. Quando se escreve uma função que retorna conjunto, o resultado desta função pode ser salvo na estrutura `FuncCallContext` — deve ser utilizado o campo `tuple_desc` ou `attinmeta`, respectivamente.

Quando se trabalha com `Datum` deve ser utilizada

```
HeapTuple heap_formtuple(TupleDesc tupdesc, Datum *values, char *nulls)
```

para construir `HeapTuple` a partir dos dados do usuário na forma de `Datum`.

Quando se trabalha com cadeias de caracteres C deve ser utilizada

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)
```

para construir `HeapTuple` a partir dos dados do usuário na forma de cadeias de caracteres C. `values` é uma matriz de cadeias de caracteres C, uma para cada atributo da linha retornada. Cada uma das cadeias de caracteres C deve estar na forma esperada pela função de entrada do tipo de dado do atributo. Para ser retornado o valor nulo para um dos atributos o ponteiro correspondente na matriz `values` deve estar definido como `NULL`. Esta função precisa ser chamada novamente para cada linha retornada.

Uma vez que se tenha construído a tupla a ser retornada pela função, esta tupla deve ser convertida no tipo `Datum`. Deve ser utilizada a função

```
HeapTupleGetDatum(HeapTuple tuple)
```

para converter `HeapTuple` em um `Datum` válido. Este `Datum` pode ser retornado diretamente se a intenção for retornar apenas uma única linha, ou pode ser utilizado como o valor retornado corrente em uma função que retorna conjunto.

Na próxima seção é mostrado um exemplo.

### 31.9.10. Retorno de conjunto a partir de funções na linguagem C

Existe, também, uma API especial que fornece suporte a retorno de conjuntos (várias linhas) a partir de função na linguagem C. Uma função que retorna conjunto deve seguir as convenções de chamada versão-1. Além disso, os arquivos fonte devem incluir `funcapi.h`, conforme mostrado acima.

Uma função que retorna conjunto (SRF) é chamada uma vez para cada item retornado. A SRF deve, portanto, salvar as informações de estado necessárias para se lembrar do que estava fazendo e retornar o próximo item a cada chamada. É fornecida a estrutura `FuncCallContext` para ajudar a controlar este processo. Dentro da função, é utilizado `fcinfo->flinfo->fn_extra` para manter um ponteiro para `FuncCallContext` entre as chamadas.

```
typedef struct
{
    /*
     * Número de vezes que foi chamada anteriormente
     */
    * call_cntr é inicializada com 0 por SRF_FIRSTCALL_INIT(), e
    * incrementada cada vez que SRF_RETURN_NEXT() é chamada.
    */
    uint32 call_cntr;
```

```

/*
 * OPCIONAL número máximo de chamadas
 *
 * max_calls existe apenas por comodidade, e sua definição é opcional.
 * Se não for definida, deve ser fornecida uma forma alternativa
 * para saber quando a função terminou.
 */
uint32 max_calls;

/*
 * OPCIONAL ponteiro para o encaixe (slot) do resultado
 *
 * Isto está obsoleto e somente está presente por motivo de compatibilidade,
 * ou seja, funções que retornam conjunto que utilizam a função em
 * obsolescência TupleDescGetSlot().
 */
TupleTableSlot *slot;

/*
 * OPCIONAL ponteiro para informações diversas fornecidas pelo usuário
 *
 * user_fctx é utilizado como ponteiro para os dados do usuário para
 * reter informações arbitrárias de contexto entre chamadas à função.
 */
void *user_fctx;

/*
 * OPCIONAL ponteiro para a estrutura contendo metadados do tipo do
 * atributo de entrada
 *
 * attinmeta é utilizado ao se retornar tuplas (ou seja, tipos de dado
 * compostos), não sendo usado para retornar tipos de dado base.
 * Somente é necessário quando há intenção de usar BuildTupleFromCStrings()
 * para criar a tupla a ser retornada.
 */
AttInMetadata *attinmeta;

/*
 * contexto de memória utilizado por estruturas que devem permanecer
 * existindo por várias chamadas
 *
 * multi_call_memory_ctx é definido por SRF_FIRSTCALL_INIT() e utilizado
 * por SRF_RETURN_DONE() para limpeza. É o contexto de memória mais
 * apropriado para qualquer memória a ser reutilizada entre várias chamadas
 * à SRF.
 */
MemoryContext multi_call_memory_ctx;

/*
 * OPCIONAL ponteiro para a estrutura que contém a descrição da tupla
 *
 * tuple_desc é utilizado ao se retornar tuplas (ou seja, tipos de dado
 * compostos), e somente é necessário se as tuplas forem ser construídas
 * utilizando heap_fortuple() em vez de BuildTupleFromCStrings().
 * Deve ser observado que o ponteiro armazenado aqui geralmente deve
 * ser processado primeiro por BlessTupleDesc().
 */
TupleDesc tuple_desc;
} FuncCallContext;

```

Uma SRF utiliza várias funções e macros que manipulam, automaticamente, a estrutura `FuncCallContext` (e esperam encontrá-la via `fn_extra`). Deve ser utilizado

```
SRF_IS_FIRSTCALL()
```

para determinar se a função está sendo chamada pela primeira vez, ou se está sendo chamada novamente. Na primeira chamada (apenas) deve ser utilizado

```
SRF_FIRSTCALL_INIT()
```

para inicializar `FuncCallContext`. Em todas as chamadas à função, incluindo a primeira, deve ser utilizado

```
SRF_PERCALL_SETUP()
```

para configurar de forma apropriada o uso de `FuncCallContext`, e limpar os dados retornados deixados pela passagem anterior.

Se a função tiver dados a serem retornados, deve ser utilizado

```
SRF_RETURN_NEXT(funcctx, resultado)
```

para retornar a quem chamou (`resultado` deve ser do tipo `Datum`, tanto para um único valor quanto para uma tupla preparada conforme descrito acima). Por fim, quando a função terminar de retornar os dados, deve ser utilizado

```
SRF_RETURN_DONE(funcctx)
```

para limpar e terminar a SRF.

O contexto de memória corrente quando a SRF é chamada é um contexto transiente que é limpo entre as chamadas. Isto significa que não é necessário chamar `pfree` para tudo que foi alocado usando `palloc`; vai desaparecer de qualquer forma. Entretanto, se for desejado alocar estruturas de dados que permaneçam existindo entre as chamadas, é necessário colocá-las em outro lugar. O contexto de memória referenciado por `multi_call_memory_ctx` é um local adequado para todos os dados que precisam continuar existindo até que a SRF termine sua execução. Na maioria dos casos, isto significa que é necessário trocar para `multi_call_memory_ctx` ao ser feita a configuração na primeira chamada.

Um exemplo de pseudocódigo completo se parece com o seguinte:

```
Datum
minha_funcao_retornando_conjunto(PG_FUNCTION_ARGS)
{
    FuncCallContext *funcctx;
    Datum           resultado;
    MemoryContext    contexto_antigo;
    mais declarações que se fizerem necessárias

    if (SRF_IS_FIRSTCALL())
    {
        funcctx = SRF_FIRSTCALL_INIT();
        contexto_antigo = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
        /* O código de configuração de uma única vez aparece aqui: */
        código do usuário
        se retorna tipo composto
            preparar TupleDesc e, talvez, AttInMetadata
        fim se retorna tipo composto
        código do usuário
        MemoryContextSwitchTo(contexto_antigo);
    }

    /* O código de configuração para cada uma das chamadas aparece aqui: */
    código do usuário
    funcctx = SRF_PERCALL_SETUP();
    código do usuário
```

```

/* esta é apenas uma forma possível de testar se terminou: */
if (funcctx->call_cntr < funcctx->max_calls)
{
    /* Aqui se deseja retornar outro item: */
    código do usuário
    obter o Datum resultado
    SRF_RETURN_NEXT(funcctx, resultado);
}
else
{
    /* Aqui terminou o retorno de itens e só é necessário fazer a limpeza: */
    código do usuário
    SRF_RETURN_DONE(funcctx);
}
}

```

Um exemplo completo de uma SRF simples retornando um tipo composto se parece com:

```
PG_FUNCTION_INFO_V1(teste_de_passado_por_valor);
```

Datum

```

teste_de_passado_por_valor(PG_FUNCTION_ARGS)
{
    FuncCallContext    *funcctx;
    int                call_cntr;
    int                max_calls;
    TupleDesc          tupdesc;
    AttInMetadata      *attinmeta;

    /* código executado apenas na primeira chamada a esta função */
    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext    contexto_antigo;

        /* criar o contexto da função para persistência entre chamadas */
        funcctx = SRF_FIRSTCALL_INIT();

        /* mudar para o contexto de memória apropriado para várias chamadas à função */
        contexto_antigo = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

        /* número total de tuplas a serem retornadas */
        funcctx->max_calls = PG_GETARG_UINT32(0);

        /* construir a descrição de tupla para a tupla __teste_de_passado_por_valor */
        tupdesc = RelationNameGetTupleDesc("__teste_de_passado_por_valor");

        /*
         * gerar metadados de atributos necessários posteriormente
         * para produzir tuplas a partir de cadeias de caractere C
         */
        attinmeta = TupleDescGetAttInMetadata(tupdesc);
        funcctx->attinmeta = attinmeta;

        MemoryContextSwitchTo(contexto_antigo);
    }

    /* código executado em toda chamada a esta função */
    funcctx = SRF_PERCALL_SETUP();

    call_cntr = funcctx->call_cntr;
    max_calls = funcctx->max_calls;
    attinmeta = funcctx->attinmeta;
}

```

```

if (call_cntr < max_calls)      /* fazer quando há mais a ser enviado */
{
    char          **values;
    HeapTuple     tuple;
    Datum         resultado;

    /*
     * Preparar a matriz de valores para construir a tupla retornada.
     * Deve ser uma matriz de cadeias de caracteres C a serem processadas
     * posteriormente pelas funções de entrada dos tipos.
     */
    values = (char **) palloc(3 * sizeof(char *));
    values[0] = (char *) palloc(16 * sizeof(char));
    values[1] = (char *) palloc(16 * sizeof(char));
    values[2] = (char *) palloc(16 * sizeof(char));

    snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
    snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
    snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

    /* construir a tupla */
    tuple = BuildTupleFromCStrings(atts, values);

    /* colocar a tupla dentro do datum */
    resultado = HeapTupleGetDatum(tuple);

    /* limpeza (não é realmente necessário) */
    pfree(values[0]);
    pfree(values[1]);
    pfree(values[2]);
    pfree(values);

    SRF_RETURN_NEXT(funcctx, resultado);
}
else      /* fazer quando não há mais nada a ser feito */
{
    SRF_RETURN_DONE(funcctx);
}
}

```

O código SQL para declarar esta função é:

```

CREATE TYPE __teste_de_passado_por_valor AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION teste_de_passado_por_valor(integer, integer)
    RETURNS SETOF __teste_de_passado_por_valor
    AS 'nome_do_arquivo', 'teste_de_passado_por_valor'
    LANGUAGE C IMMUTABLE STRICT;

```

Na distribuição do código fonte, o diretório contrib/tablefunc contém mais exemplos de funções que retornam conjunto.

### 31.9.11. Tipos polimórficos em argumentos e retorno

As funções na linguagem C podem ser declaradas como recebendo e retornando os tipos polimórficos `anyelement` e `anyarray`. Para obter uma explicação mais detalhada das funções polimórficas deve ser vista Seção 31.2.5. Quando o tipo do argumento da função ou do valor retornado é declarado como polimórfico, o autor da função não pode saber antecipadamente com que tipo de dado a função será chamada, ou usado no retorno. Existem duas rotinas fornecidas em `fmgr.h` que permitem uma função versão-1 descobrir o tipo de dado verdadeiro de seus argumentos, e o tipo de dado esperado no retorno. As rotinas se chamam `get_fn_expr_rettype(FmgrInfo *flinfo)` e `get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)`. Retornam o OID do tipo do argumento ou do resultado,

ou InvalidOid se a informação não estiver disponível. A estrutura flinfo normalmente é acessada por fcinfo->flinfo. O parâmetro argnum começa por zero.

Por exemplo, suponha que se deseje escrever uma função que aceite um único elemento de qualquer tipo, e retorne uma matriz unidimensional deste tipo:

```
PG_FUNCTION_INFO_V1(constroi_matriz);
Datum
constroi_matriz(PG_FUNCTION_ARGS)
{
    ArrayType *resultado;
    Oid        tipo_do_elemento = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum      elemento;
    int16      typlen;
    bool       typbyval;
    char       typalign;
    int        ndims;
    int        dims[MAXDIM];
    int        lbs[MAXDIM];

    if (!OidIsValid(tipo_do_elemento))
        elog(ERROR, "não foi possível determinar o tipo de dado de entrada");

    /* obter o elemento fornecido */
    element = PG_GETARG_DATUM(0);

    /* temos uma dimensão */
    ndims = 1;
    /* e um elemento */
    dims[0] = 1;
    /* e o limite inferior é 1 */
    lbs[0] = 1;

    /* obter as informações requeridas sobre o tipo do elemento */
    get_typlenbyvalalign(tipo_do_elemento, &typlen, &typbyval, &typalign);

    /* construir a matriz */
    resultado = construct_md_array(&elemento, ndims, dims, lbs,
                                   tipo_do_elemento, typlen, typbyval, typalign);

    PG_RETURN_ARRAYTYPE_P(resultado);
}
```

O seguinte comando declara a função constroi\_matriz no SQL:

```
CREATE FUNCTION constroi_matriz(anyelement) RETURNS anyarray
AS 'DIRETÓRIO/funcs', 'constroi_matriz'
LANGUAGE C STRICT;
```

Deve ser observada a utilização de STRICT; isto é essencial uma vez que código não se importa em testar entrada nula.

### 31.9.12. Exemplos de funções escritas em C

**Nota:** Esta seção foi escrita pelo tradutor, não fazendo parte do manual original.

Nos exemplos desta seção foram utilizados os arquivos: funcoes.c (./funcoes.c) para as funções codificadas na linguagem C; funcoes.sql (./funcoes.sql) para os comandos SQL; e o arquivo Makefile

```
MODULES = funcoes
DATA = funcoes.sql

PGXS := $(shell pg_config --pgxs)
include $(PGXS)
```

Todos colocados no diretório ~/funcoes. O código fonte da distribuição do PostgreSQL foi descompactado sob o diretório ~/postgresql-8.0.0 e feita a instalação. Com isto, a geração do arquivo de biblioteca compartilhada foi feita usando os comandos:

```
-- Gerar a biblioteca compartilhada
cd ~/funcoes
make
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement \
    -Wold-style-definition -Wendif-labels -fno-strict-aliasing \
    -fpic -I. -I/usr/local/pgsql/include/server \
    -I/usr/local/pgsql/include/internal -D_GNU_SOURCE \
    -c -o funcoes.o funcoes.c
gcc -shared -o funcoes.so funcoes.o
rm funcoes.o
make install
mkdir -p -- /usr/local/pgsql/share/contrib
/bin/sh /usr/local/pgsql/lib/pgxs/src/makefiles/../../config/install-sh \
    -c -m 644 ./funcoes.sql /usr/local/pgsql/share/contrib
/bin/sh /usr/local/pgsql/lib/pgxs/src/makefiles/../../config/install-sh \
    -c -m 755 funcoes.so /usr/local/pgsql/lib
```

### Exemplo 31-1. Funções C para cálculo de dígitos verificadores- FEBRABAN

Neste exemplo são mostradas duas funções para cálculo do dígito verificador. A primeira função calcula o dígito verificador módulo 11 e a segunda módulo 10. As duas funções são sobrecarregadas (consulte a Seção 31.5): recebem um parâmetro, o número, e retornam o dígito verificador; ou recebem dois parâmetros, o número e o dígito verificador, e retornam verdade ou falso caso o dígito esteja correto ou errado, respectivamente.

O cálculo do dígito verificador módulo 11 é feito da seguinte forma: os dígitos são multiplicados da direita para a esquerda pela seqüência de 2 a 9, ou seja, por 2, 3, 4, 5, 6, 7, 8, 9, 2, 3, 4, e assim por diante; os resultados do produto dos dígitos pelos valores da seqüência são somados; a soma é dividida por 11, e o resto da divisão é obtido; o dígito verificador é calculado subtraindo o resto de 11; quando o resto for igual a 0 ou 1 o dígito verificador será igual a 0. Segundo o Manual Técnico Operacional - Bloquetos de Cobrança - FEBRABAN ([http://www.bradesco.com.br/br/pj/conteudo/sol\\_rec/pdf/manualtecnico.pdf](http://www.bradesco.com.br/br/pj/conteudo/sol_rec/pdf/manualtecnico.pdf)), deve ser utilizado o dígito 1 quando o resto for 0, 10 ou 1. Existem Secretarias de Fazenda onde no cálculo do dígito verificador a seqüência retorna a 2 antes de chegar a 9. Tome cuidado antes de utilizar esta função.

O cálculo do dígito verificador módulo 10 é feito da seguinte forma: os dígitos são multiplicados da direita para a esquerda pela seqüência 2, 1, 2, 1, e assim por diante; os “algarismos” dos resultados do produto dos dígitos pelos valores da seqüência são somados individualmente; a soma é dividida por 10, e o resto da divisão é obtido; o dígito verificador é calculado subtraindo o resto de 10; quando o resto for igual a 0 o dígito verificador será igual a 0. Fonte: Manual Técnico Operacional - Bloquetos de Cobrança - FEBRABAN. Existe outra forma de calcular este dígito verificador onde os resultados dos produtos, e não os dígitos dos resultados dos produtos, são somados. Tome cuidado antes de utilizar esta função.

Este exemplo mostra o código fonte C da biblioteca compartilhada para cálculo de dígitos verificadores conforme o Manual Técnico Operacional - Bloquetos de Cobrança - FEBRABAN.

```
/*
 * Rotina para cálculo do dígito verificador módulo 11.
 * Recebe dois argumentos, número e dígito verificador na forma de caracteres.
 * Retorna verdade se o dígito verificador estiver correto, falso caso
 * contrário, ou nulo em caso de erro.
 */

PG_FUNCTION_INFO_V1(dv11);

Datum
dv11(PG_FUNCTION_ARGS) {
    int    digito = 0,      // Dígito verificador
           fator  = 2;      // Fator de multiplicação
    text *num;              // Primeiro argumento = número
```

```

text *dv;          // Segundo argumento = dígito verificador
char *c;           // Ponteiro para os caracteres do número
/* Verificar o recebimento de argumento nulo */
if (PG_ARGISNULL(0) || PG_ARGISNULL(1))
    PG_RETURN_NULL();
/* Receber os argumentos */
num = PG_GETARG_TEXT_P(0);
dv = PG_GETARG_TEXT_P(1);
/* Verificar o recebimento de argumento vazio */
if ((VARSIZE(num) == VARHDRSZ) || (VARSIZE(dv) == VARHDRSZ)) {
    PG_RETURN_NULL();
}
/* Verificar dígito verificador não dígito */
if (!isdigit(*VARDATA(dv))) PG_RETURN_NULL();
/* Calcular o dígito verificador */
for (c = VAREND(num); c >= VARDATA(num); c--) {
    if (!isdigit(*c)) PG_RETURN_NULL(); // Verificar se é dígito
    digito += VALOR(*c) * fator;
    if (++fator > 9) fator = 2;
}
digito = 11 - ( digito % 11 );
if (digito >= 10) digito = 0; // Restos 0 ou 1 dígito = 0
// Retornar verdade ou falso
PG_RETURN_BOOL (digito == VALOR(*VARDATA(dv)));
}

/*
 * Rotina para cálculo do dígito verificador módulo 11.
 * Recebe um argumento, o número na forma de caracteres.
 * Retorna o dígito verificador, ou nulo em caso de erro.
 */

PG_FUNCTION_INFO_V1(dv11dig);

Datum
dv11dig(PG_FUNCTION_ARGS) {
    int  digito=0,      // Dígito verificador
        fator=2;       // Fator de multiplicação
    text *num;          // Único argumento = número
    char *c;            // Ponteiro para os caracteres do número
    int32 tamanho;      // Tamanho do resultado da função
    text *resultado;    // Ponteiro para o resultado da função
    /* Verificar o recebimento de argumento nulo */
    if (PG_ARGISNULL(0))
        PG_RETURN_NULL();
    /* Receber o argumento */
    num = PG_GETARG_TEXT_P(0);
    /* Verificar o recebimento de argumento vazio */
    if (VARSIZE(num) == VARHDRSZ) {
        PG_RETURN_NULL();
    }
    /* Calcular o dígito verificador */
    for (c = VAREND(num); c >= VARDATA(num); c--) {
        if (!isdigit(*c)) PG_RETURN_NULL(); // Verificar se é dígito
        digito += VALOR(*c) * fator;
        if (++fator > 9) fator = 2;
    }
    digito = 11 - ( digito % 11 );
    if (digito >= 10) digito = 0; // Restos 0 ou 1 dígito = 0
    /* Retornar o dígito verificador */
    tamanho = VARHDRSZ + sizeof(char);
    resultado = (text *) palloc(tamanho);

```



```

memset((void *) resultado, 0, tamanho);
VARATT_SIZEP(resultado) = tamanho;
*VARDATA(resultado) = (char) DIGITO(digito);
PG_RETURN_TEXT_P(resultado);
}

/*
 * Rotina para cálculo do dígito verificador módulo 10 - FEBRABAN
 * Recebe dois argumentos, número e dígito verificador na forma de caracteres.
 * Retorna verdade se o dígito verificador estiver correto, falso caso
 * contrário, ou nulo em caso de erro.
 */

PG_FUNCTION_INFO_V1(dv10);

Datum
dv10(PG_FUNCTION_ARGS) {
    int    digito = 0,    // Dígito verificador
           fator   = 2,    // Fator de multiplicação
           produto;        // Produto = dígito x fator
    text *num;            // Primeiro argumento = número
    text *dv;             // Segundo argumento = dígito verificador
    char *c;              // Ponteiro para os caracteres do número
    /* Verificar o recebimento de argumento nulo */
    if (PG_ARGISNULL(0) || PG_ARGISNULL(1))
        PG_RETURN_NULL();
    /* Receber os argumentos */
    num = PG_GETARG_TEXT_P(0);
    dv  = PG_GETARG_TEXT_P(1);
    /* Verificar o recebimento de argumento vazio */
    if ((VARSIZE(num) == VARHDRSZ) || (VARSIZE(dv) == VARHDRSZ)) {
        PG_RETURN_NULL();
    }
    /* Verificar dígito verificador não dígito */
    if (!isdigit(*VARDATA(dv))) PG_RETURN_NULL();
    /* Calcular o dígito verificador */
    for (c = VAREND(num); c >= VARDATA(num); c--) {
        if (!isdigit(*c)) PG_RETURN_NULL(); // Verificar se é dígito
        produto = VALOR(*c) * fator;
        digito += produto/10 + produto%10;
        if (--fator < 1) fator = 2;
    }
    digito = 10 - ( digito % 10 );
    if (digito == 10) digito = 0;
    /* Retornar verdade ou falso */
    PG_RETURN_BOOL (digito == VALOR(*VARDATA(dv)));
}

/*
 * Rotina para cálculo do dígito verificador módulo 10 - FEBRABAN.
 * Recebe um argumento, o número na forma de caracteres.
 * Retorna o dígito verificador, ou nulo em caso de erro.
 */

PG_FUNCTION_INFO_V1(dv10dig);

Datum
dv10dig(PG_FUNCTION_ARGS) {
    int    digito = 0,    // Dígito verificador
           fator   = 2,    // Fator de multiplicação
           produto;        // Produto = dígito x fator
    text *num;            // Único argumento = número

```

```

char *c;           // Ponteiro para os caracteres do número
int32 tamanho;     // Tamanho do resultado da função
text *resultado;   // Ponteiro para o resultado da função
/* Verificar o recebimento de argumento nulo */
if (PG_ARGISNULL(0))
    PG_RETURN_NULL();
/* Receber o argumento */
num = PG_GETARG_TEXT_P(0);
/* Verificar o recebimento de argumento vazio */
if (VARSIZE(num) == VARHDRSZ) {
    PG_RETURN_NULL();
}
/* Calcular o dígito verificador */
for (c = VAREND(num); c >= VARDATA(num); c--) {
    if (!isdigit(*c)) PG_RETURN_NULL(); // Verificar se é dígito
    produto = VALOR(*c) * fator;
    digito += produto/10 + produto%10;
    if (--fator < 1) fator = 2;
}
digito = 10 - ( digito % 10 );
if (digito == 10) digito = 0;
/* Retornar o dígito verificador */
tamanho = VARHDRSZ + sizeof(char);
resultado = (text *) palloc(tamanho);
memset((void *) resultado, 0, tamanho);
VARATT_SIZEP(resultado) = tamanho;
*VARDATA(resultado) = (char) DIGITO(digito);
PG_RETURN_TEXT_P(resultado);
}

```

O código SQL para declarar estas funções é:

```

LOAD 'funcoes';

CREATE FUNCTION dv11(text,text) RETURNS boolean
AS '$libdir/funcoes', 'dv11'
LANGUAGE C STRICT;

CREATE FUNCTION dv11(text) RETURNS text
AS '$libdir/funcoes', 'dv11dig'
LANGUAGE C STRICT;

CREATE FUNCTION dv10(text,text) RETURNS boolean
AS '$libdir/funcoes', 'dv10'
LANGUAGE C STRICT;

CREATE FUNCTION dv10(text) RETURNS text
AS '$libdir/funcoes', 'dv10dig'
LANGUAGE C STRICT;

```

Utilização das funções com informações retiradas do código de barras de bloquitos de cobrança bancária:

```
SELECT dv11('3419112820459284738210122301001623770000034439','7');
```

```

dv11
-----
t
(1 linha)

```

```
SELECT dv11('3419112820459284738210122301001623770000034439');
```

```

dv11
-----
7

```

```
(1 linha)

SELECT dv10('0063504142','9');

dv10
-----
t
(1 linha)

SELECT dv10('0063504142');

dv10
-----
9
(1 linha)
```

### Exemplo 31-2. Funções C para validar o número do CPF e do CNPJ

Neste exemplo são mostradas funções para validar o número do CPF e do CNPJ. A primeira função recebe o número do CPF com 8 a 11 dígitos, e a segunda recebe o número do CNPJ com 14 dígitos. Ambas retornam o valor booleano verdade se os dígitos verificadores estiverem corretos, ou falso caso contrário. Se o argumento for nulo, não tiver o número de dígitos esperado, ou contiver um dígito não numérico, as funções retornam nulo.

```
/*
 * Rotina para validação do CPF.
 * Recebe um argumento, o número do CPF com oito a onze dígitos.
 * Retorna verdade se os dígitos verificadores do CPF estiverem corretos,
 * falso caso contrário, ou nulo se o argumento for nulo, não tiver entre
 * 8 e 11 dígitos, ou contiver um dígito não numérico. Não serão considerados
 * válidos os seguintes CPF: 000.000.000-00, 111.111.111-11, 222.222.222-22,
 * 333.333.333-33, 444.444.444-44, 555.555.555-55, 666.666.666-66,
 * 777.777.777-77, 888.888.888-88, 999.999.999-99.
 */

PG_FUNCTION_INFO_V1(cpf);

Datum
cpf(PG_FUNCTION_ARGS) {
    text *num;          // Único argumento = número do CPF
    bool iguais;         // Todos os dígitos são iguais
    int fator,          // Fator de multiplicação
        digito;         // Dígito verificador
    char *cpf;          // Número do CPF com 11 dígitos
    char *c;            // Ponteiro para os caracteres do CPF
    /* Verificar o recebimento de argumento nulo */
    if (PG_ARGISNULL(0))
        PG_RETURN_NULL();
    /* Receber o argumento */
    num = PG_GETARG_TEXT_P(0);
    /* Verificar se o CPF tem entre 8 e 11 dígitos */
    if ( ((VARSIZE(num) - VARHDRSZ) > 11*sizeof(char)) ||
         ((VARSIZE(num) - VARHDRSZ) < 8*sizeof(char))
       ) PG_RETURN_NULL();
    /* CPF com 11 dígitos */
    cpf = (char *) palloc(11*sizeof(char));          // Reservar memória
    strncpy (cpf, "00000000000", 11*sizeof(char));   // Preencher com zeros
    memcpy (cpf+11*sizeof(char)-(VARSIZE(num)-VARHDRSZ), // Destino
            VARDATA(num),                             // Origem
            VARSIZE(num)-VARHDRSZ);                   // Comprimento
    /* Verificar se todos os dígitos são iguais */
    iguais = true;
    for (c=cpf; c<cpf+9*sizeof(char); *c++) {
        if (*c != *(c+sizeof(char))) {
```

```

        iguais = false;
        break;
    }
}
if (iguais) PG_RETURN_BOOL(false);
/* Validar o primeiro dígito verificador */
for (c=cpf, digito=0, fator=10; fator>=2; fator--) {
    if (!isdigit(*c)) PG_RETURN_NULL(); // Retornar nulo se não for dígito
    digito += VALOR(*c++) * fator;
}
digito = 11 - ( digito % 11 );
if (digito >= 10) digito = 0; // Restos 0 ou 1 digito = 0
/* Retornar nulo se o primeiro dígito verificador não for um dígito */
if (!isdigit(*c)) PG_RETURN_NULL();
// Retornar falso se o primeiro dígito não estiver correto
if (digito != VALOR(*c)) PG_RETURN_BOOL(false);
/* Validar o segundo dígito verificador */
for (c=cpf, digito=0, fator=11; fator>=2; fator--) {
    if (!isdigit(*c)) PG_RETURN_NULL(); // Retornar nulo se não for dígito
    digito += VALOR(*c++) * fator;
}
digito = 11 - ( digito % 11 );
if (digito >= 10) digito = 0; // Restos 0 ou 1 digito = 0
/* Retornar nulo se o segundo dígito verificador não for um dígito */
if (!isdigit(*c)) PG_RETURN_NULL();
// Retornar verdade ou falso de acordo com o segundo dígito verificador
PG_RETURN_BOOL (digito == VALOR(*c));
}

/*
 * Rotina para validação do CNPJ.
 * Recebe um argumento, o número do CNPJ com quatorze dígitos.
 * Retorna verdade se os dígitos verificadores do CNPJ estiverem corretos,
 * falso caso contrário, ou nulo se o argumento for nulo, não tiver 14 dígitos,
 * ou contiver um dígito não numérico.
 */

PG_FUNCTION_INFO_V1(cnpj);

Datum
cnpj(PG_FUNCTION_ARGS) {
    text *num;          // Único argumento = número do CNPJ
    int  fator,          // Fator de multiplicação
        digito;          // Dígito verificador
    char *c;             // Ponteiro para os caracteres do CNPJ
    /* Verificar o recebimento de argumento nulo */
    if (PG_ARGISNULL(0))
        PG_RETURN_NULL();
    /* Receber o argumento */
    num = PG_GETARG_TEXT_P(0);
    /* Verificar se o CNPJ tem 14 dígitos */
    if ( (VARSIZE(num) - VARHDRSZ) != 14*sizeof(char) ) {
        PG_RETURN_NULL();
    }
    /* Validar o primeiro dígito verificador */
    for (c=VARDATA(num), digito=0, fator=13; fator>=2; fator--) {
        if (!isdigit(*c)) PG_RETURN_NULL(); // Retornar nulo se não for dígito
        digito += VALOR(*c++) * (fator>9 ? fator-8 : fator);
    }
    digito = 11 - ( digito % 11 );
    if (digito >= 10) digito = 0; // Restos 0 ou 1 digito = 0
    // Retornar falso se o primeiro dígito não estiver correto

```

```

if (digito != VALOR(*c)) PG_RETURN_BOOL(false);
/* Validar o segundo dígito verificador */
for (c=VARDATA(num), digito=0, fator=14; fator>=2; fator--) {
    if (!isdigit(*c)) PG_RETURN_NULL(); // Retornar nulo se não for dígito
    digito += VALOR(*c++) * (fator>9 ? fator-8 : fator);
}
digito = 11 - ( digito % 11 );
if (digito >= 10) digito = 0; // Restos 0 ou 1 digito = 0
// Retornar verdade ou falso de acordo com o segundo dígito verificador
PG_RETURN_BOOL (digito == VALOR(*c));
}

```

O código SQL para declarar estas funções é:

```

LOAD 'funcoes';

CREATE FUNCTION cpf(text) RETURNS boolean
AS '$libdir/funcoes', 'cpf'
LANGUAGE C STRICT;

CREATE FUNCTION cnpj(text) RETURNS boolean
AS '$libdir/funcoes', 'cnpj'
LANGUAGE C STRICT;

```

Os números de CPF utilizados para testar a função não estão divulgados por motivo de confidencialidade. Os números de CNPJ usados para testar a função são de órgãos públicos.

```
SELECT cnpj('42498634000166');
```

```

cnpj
-----
t
(1 linha)

```

```
SELECT cnpj('42498733000148');
```

```

cnpj
-----
t
(1 linha)

```

### Exemplo 31-3. Função C para validar o número de inscrição eleitoral

Neste exemplo é mostrada uma função para validar o número de inscrição eleitoral. A função recebe como parâmetro o número de inscrição eleitoral, e retorna o valor booleano verdade se os dígitos verificadores estiverem corretos, ou falso caso contrário. Se o argumento for nulo, não tiver o número de dígitos esperado, ou contiver um dígito não numérico, a função retorna nulo. A função não verifica se a unidade da federação é válida.

A função é baseada na Resolução nº 20.132, de 19.03.98, do Tribunal Superior Eleitoral, art. 10, cujo texto é reproduzido abaixo:

Art. 10 - Os Tribunais Regionais Eleitorais farão distribuir, observada a seqüência numérica fornecida pela Secretaria de Informática, às Zonas Eleitorais da respectiva Circunscrição, séries de números de inscrição eleitoral, a serem utilizados na forma deste artigo.

Parágrafo único - O número de inscrição compor-se-á de até 12 (doze) algarismos, por Unidade da Federação, assim discriminados:

- a) os 8 (oito) primeiros algarismos serão seqüenciados, desprezando-se, na emissão, os zeros à esquerda;
- b) os 2 (dois) algarismos seguintes serão representativos da Unidade da Federação de origem da inscrição, conforme códigos constantes da seguinte tabela:

01 - São Paulo; 02 - Minas Gerais; 03 - Rio de Janeiro; 04 - Rio Grande do Sul; 05 - Bahia; 06 - Paraná; 07 - Ceará; 08 - Pernambuco; 09 - Santa Catarina; 10 - Goiás; 11 - Maranhão; 12 - Paraíba; 13 - Pará; 14 - Espírito Santo; 15 - Piauí; 16 -

Rio Grande do Norte; 17 - Alagoas; 18 - Mato Grosso; 19 - Mato Grosso do Sul; 20 - Distrito Federal; 21 - Sergipe; 22 - Amazonas; 23 - Rondônia; 24 - Acre; 25 - Amapá; 26 - Roraima; 27 - Tocantins; 28 - Exterior (ZZ).

c) os 2 (dois) últimos algarismos constituirão dígitos verificadores, determinados com base no módulo 11 (onze), sendo o primeiro calculado sobre o número sequencial e o último sobre o código da Unidade da Federação seguido do primeiro dígito verificador.

Código fonte da função:

```
/*
 * Rotina para validação do número de inscrição eleitoral.
 * Recebe um argumento, o número de inscrição eleitoral com doze dígitos.
 * Retorna verdade se os dígitos verificadores do número de inscrição
 * eleitoral estiverem corretos, falso caso contrário, ou nulo se o argumento
 * for nulo, não tiver 12 dígitos, ou contiver um dígito não numérico.
 */

PG_FUNCTION_INFO_V1(nie);

Datum
nie(PG_FUNCTION_ARGS) {
    text *num;          // Único argumento = número de inscrição eleitoral
    int  fator,          // Fator de multiplicação
        digito;          // Dígito verificador
    char *nie;           // Número do inscrição eleitoral com 12 dígitos
    char *c;             // Ponteiro para os caracteres do número de inscrição
    /* Verificar o recebimento de argumento nulo */
    if (PG_ARGISNULL(0))
        PG_RETURN_NULL();
    /* Receber o argumento */
    num = PG_GETARG_TEXT_P(0);
    /* Verificar se o número de inscrição tem entre 10 e 12 dígitos */
    if ( ((VARSIZE(num) - VARHDRSZ) > 12*sizeof(char)) ||
         ((VARSIZE(num) - VARHDRSZ) < 10*sizeof(char))
        ) PG_RETURN_NULL();
    /* Número de inscrição eleitoral com 12 dígitos */
    nie = (char *) palloc(12*sizeof(char));          // Reservar memória
    strncpy (nie, "000000000000", 12*sizeof(char));  // Preencher com zeros
    memcpy (nie+12*sizeof(char)-(VARSIZE(num)-VARHDRSZ), // Destino
            VARDATA(num),                               // Origem
            VARSIZE(num)-VARHDRSZ);                     // Comprimento
    /* Validar o primeiro dígito verificador */
    for (c=nie, digito=0, fator=9; fator>=2; fator--) {
        if (!isdigit(*c)) PG_RETURN_NULL(); // Retornar nulo se não for dígito
        digito += VALOR(*c++) * fator;
    }
    digito = 11 - ( digito % 11 );
    if (digito >= 10) digito = 0; // Restos 0 ou 1 digito = 0
    // Retornar falso se o primeiro dígito não estiver correto
    if (digito != VALOR(*(c+2*sizeof(char)))) PG_RETURN_BOOL(false);
    /* Validar o segundo dígito verificador */
    for (digito=0, fator=4; fator>=2; fator--) {
        if (!isdigit(*c)) PG_RETURN_NULL(); // Retornar nulo se não for dígito
        digito += VALOR(*c++) * fator;
    }
    digito = 11 - ( digito % 11 );
    if (digito >= 10) digito = 0; // Restos 0 ou 1 digito = 0
    // Retornar verdade ou falso de acordo com o segundo dígito verificador
    PG_RETURN_BOOL (digito == VALOR(*c));
}
```

O código SQL para declarar esta função é:

```
LOAD 'funcoes';
```

```
CREATE FUNCTION nie(text) RETURNS boolean
AS '$libdir/funcoes', 'nie'
LANGUAGE C STRICT;
```

Os números de inscrição eleitoral utilizados para testar a função não estão divulgados por motivo de confidencialidade.

#### Exemplo 31-4. Função C para calcular o Máximo Divisor Comum

Neste exemplo é mostrada uma função para calcular o máximo divisor comum (MDC) entre dois números inteiros, utilizando o Algoritmo Euclidiano.

```
/*
 * Rotina para cálculo do Máximo Divisor Comum (MDC).
 * Utiliza o Algoritmo de Euclides ou Algoritmo Euclidiano.
 * Recebe com parâmetro dois números inteiros e retorna o MDC.
 * Fonte: http://www2.fundao.pro.br/articles.asp?cod=151
 */

PG_FUNCTION_INFO_V1(mdc);

Datum
mdc(PG_FUNCTION_ARGS) {
    int  a,      // Primeiro número
        b,      // Segundo número
        r,      // Resto da divisão de A por B
        t;      // Armazenamento temporário (troca)
    /* Verificar o recebimento de argumento nulo */
    if (PG_ARGISNULL(0) || PG_ARGISNULL(1))
        PG_RETURN_NULL();
    /* Receber os argumentos */
    a = PG_GETARG_INT32(0);
    b = PG_GETARG_INT32(1);
    /* Garantir que A seja o maior valor */
    if ( a < b ) {
        t = a;
        a = b;
        b = t;
    }
    /* Calcular o MDC */
    while ( b != 0 )
    {
        r = a % b;
        a = b;
        b = r;
    }
    PG_RETURN_INT32(a);
}
```

O código SQL para declarar esta função é:

```
LOAD 'funcoes';

CREATE FUNCTION mdc(int, int) RETURNS int
AS '$libdir/funcoes', 'mdc'
LANGUAGE C STRICT;
```

Exemplos de utilização da função:

```
SELECT mdc(144,1024);
```

```
mdc
-----
16
(1 linha)
```

```
SELECT mdc(8192,224);
```

```
mdc
-----
 32
(1 linha)
```

### Exemplo 31-5. Função C para concatenar duas cadeias de caracteres

Neste exemplo é mostrada uma função para concatenar duas cadeias de caracteres.

```
/*
 * Função para concatenar duas cadeias de caracteres.
 * Se encontra no arquivo src/tutorial/funcs_new.c da
 * distribuição do código fonte do PostgreSQL.
 */

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text      *arg1 = PG_GETARG_TEXT_P(0);
    text      *arg2 = PG_GETARG_TEXT_P(1);
    int32      tam_novo_texto = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text      *novo_texto = (text *) palloc(tam_novo_texto);

    memset((void *) novo_texto, 0, tam_novo_texto);
    VARATT_SIZEP(novo_texto) = tam_novo_texto;
    strncpy(VARDATA(novo_texto), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    strncat(VARDATA(novo_texto), VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(novo_texto);
}
```

O código SQL para declarar esta função é:

```
LOAD 'funcoes';

CREATE FUNCTION concat(text, text) RETURNS text
    AS '$libdir/funcoes', 'concat_text'
    LANGUAGE C STRICT;
```

Exemplos de utilização da função:

```
SELECT concat('Postgre','SQL');

concat
-----
PostgreSQL
(1 linha)

SELECT concat(concat('ae','io'),'u') AS vogais;

vogais
-----
aeiou
(1 linha)
```

### Exemplo 31-6. Listar as funções C definidas pelo usuário

A consulta abaixo lista o nome, número de argumentos e o tipo retornado por todas as funções C definidas pelo usuário.

```
SELECT n.nspname, p.proname, p.pronargs, format_type(t.oid, null) as return_type
FROM pg_namespace n, pg_proc p,
     pg_language l, pg_type t
```



```

WHERE p.pronamespace = n.oid
      and n.nspname not like 'pg\\_%'          -- sem catálogos
      and n.nspname != 'information_schema' -- sem information_schema
      and p.prolang = l.oid
      and p.prorettype = t.oid
      and l.lanname = 'c'
ORDER BY nspname, proname, pronargs, return_type;

```

nspname	praname	pronargs	return_type
public	cnpj	1	boolean
public	concat	2	text
public	cpf	1	boolean
public	dv10	1	text
public	dv10	2	boolean
public	dv11	1	text
public	dv11	2	boolean
public	mdc	2	integer
public	nie	1	boolean

(9 linhas)

## 31.10. Agregações definidas pelo usuário

As funções de agregação no PostgreSQL são expressas como funções de *valor de estado* e funções de *transição de estado*, ou seja, uma agregação pode ser definida em termos de um estado que é modificado toda vez que um item de entrada é processado. Para definir uma nova função de agregação deve ser selecionado um tipo de dado para o valor do estado, um valor inicial para o estado, e uma função de transição de estado. A função de transição de estado é uma função comum, que também poderia ser utilizada fora do contexto da agregação. No caso do resultado desejado para a agregação ser diferente do dado mantido no valor de estado sendo processado, também pode ser especificada uma *função final*.

Portanto, além dos tipos de dado do argumento e do resultado vistos pelo usuário, existe também um tipo de dado interno para o valor de estado que pode ser diferente tanto do tipo de dado do argumento, quanto do tipo de dado do resultado.

Quando se define uma agregação que não utiliza uma função final, se tem uma agregação que computa uma função processadora dos valores da coluna para cada linha. `sum` é um exemplo deste tipo de agregação. `sum` começa em zero e sempre adiciona o valor da linha corrente ao total sendo calculado. Por exemplo, se for desejado desenvolver uma agregação `sum` que trabalhe com tipo de dado para números complexos, somente é necessário adicionar uma função para este tipo de dado. A definição da agregação poderia ser:

```

CREATE AGGREGATE complex_sum (
    sfunc = complex_add,
    basetype = complex,
    stype = complex,
    initcond = '(0,0)'
);

SELECT complex_sum(a) FROM test_complex;

complex_sum
-----
(34,53.9)

```

(Na prática apenas chamamos a agregação de `sum` e confiamos que o PostgreSQL descubra que tipo de soma deve ser aplicada a uma coluna do tipo `complex`)

A definição de `sum` acima retorna zero, a condição de estado inicial, se não houver nenhum valor de entrada diferente de nulo. Pode ser que neste caso se prefira retornar nulo em vez de zero — o padrão SQL espera que a agregação `sum` se comporte desta maneira. Isto pode ser feito simplesmente omitindo a linha `initcond` para que a condição de estado inicial seja nula. Normalmente isto significaria que `sfunc` precisaria verificar uma entrada tendo nulo na condição de estado, mas para `sum`, e algumas outras agregações simples como `max` e `min`, basta atribuir o primeiro valor de entrada não nulo à variável de estado, e depois começar a aplicar a função de transição a partir do segundo valor de entrada não nulo. O

PostgreSQL faz isto automaticamente quando a condição inicial é nula, e a função de transição está marcada como “strict” (ou seja, não é chamada para entradas nulas).

Outro ponto do comportamento padrão da função de transição “strict” a ser ressaltado, é que o valor de estado anterior permanece inalterado quando é encontrado um valor de entrada nulo. Portanto, os valores nulos são ignorados. Se for desejado um comportamento diferente para os valores nulos, a função de transição não deve ser definida como estrita, e deve ser codificada testando as entradas nulas e fazendo o que for necessário.

Um exemplo mais complexo de agregação é `avg` (média). Requer dois elementos de estado de execução: a soma das entradas e o contador do número de entradas. O resultado final é obtido pela divisão destas quantidades. Usualmente a média é implementada utilizando uma matriz com dois elementos para o valor de estado. Por exemplo, a implementação interna de `avg(float8)` se parece com:

```
CREATE AGGREGATE avg (
    sfunc = float8_accum,
    basetype = float8,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0}'
);
```

As funções de agregação podem utilizar funções de transição de estado e funções finais polimórficas e, portanto, a mesma função pode ser utilizada para implementar várias agregações. Para obter informações sobre funções polimórficas deve ser consultada a Seção 31.2.5. Indo um passo adiante, a própria função de agregação pode ser especificada com um tipo base e um tipo de estado polimórficos, permitindo que uma única definição de agregação sirva para vários tipos de dado de entrada. Abaixo está um exemplo de agregação polimórfica:

```
CREATE AGGREGATE array_accum (
    sfunc = array_append,
    basetype = anyelement,
    stype = anyarray,
    initcond = '{}'
);
```

Neste caso, o tipo de dado real do estado, para qualquer chamada à agregação, é o tipo de dado da matriz que possui o tipo de dado real de entrada como seus elementos.

Abaixo estão mostradas as saídas da utilização de dois tipos de dado diferente como argumentos:

```
SELECT attrelid::regclass, array_accum(attname)
FROM pg_attribute
WHERE attnum > 0 AND attrelid = 'pg_user'::regclass
GROUP BY attrelid;
```

```
attrelid | array_accum
-----+-----
pg_user | {username,usesysid,usecreatedb,usesuper,usecatupd,passwd,valuntil,useconfig}
(1 linha)
```

```
SELECT attrelid::regclass, array_accum(atttypid)
FROM pg_attribute
WHERE attnum > 0 AND attrelid = 'pg_user'::regclass
GROUP BY attrelid;
```

```
attrelid | array_accum
-----+-----
pg_user | {19,23,16,16,16,25,702,1009}
(1 linha)
```

Para obter mais detalhes deve ser visto o comando `CREATE AGGREGATE`.

### 31.11. Tipos definidos pelo usuário

Conforme descrito na Seção 31.2, o PostgreSQL pode ser estendido para dar suporte a novos tipos de dado. Esta seção descreve como definir novos tipos base, que são tipos de dado definidos abaixo do nível da linguagem SQL. A criação de um novo tipo base requer a implementação de funções para operar o tipo em uma linguagem de baixo nível, geralmente a linguagem C.

Os exemplos desta seção podem ser encontrados nos arquivos `complex.sql` e `complex.c` no diretório `src/tutorial` da distribuição do código fonte. <sup>4</sup> Para obter informações sobre como executar os exemplos deve ser visto o arquivo `README` presente neste diretório.

Os tipos definidos pelo usuário devem possuir função de entrada e de saída, sempre. Estas funções determinam como o tipo aparece nas cadeias de caracteres (na entrada pelo usuário e na saída para o usuário), e como estes tipos ficam organizados na memória. A função de entrada recebe como argumento uma cadeia de caracteres terminada por nulo, e retorna a representação interna (em memória) do tipo. A função de saída recebe como argumento a representação interna do tipo, e retorna uma cadeia de caracteres terminada por nulo. Se for desejado fazer algo mais com o tipo que simplesmente armazená-lo, devem ser fornecidas funções adicionais para implementar as operações desejadas para o tipo.

Supondo que se deseja definir o tipo `complex` para representar os números complexos, uma forma natural de representar um número complexo na memória seria através da seguinte estrutura na linguagem C:

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

É necessário torná-lo um tipo passado por referência, uma vez que este tipo é muito grande para caber em um único valor `Datum`.

Para representação externa do tipo foi escolhida uma cadeia de caracteres na forma `(x,y)`.

Geralmente não é difícil escrever as funções de entrada e de saída, em especial a função de saída. Mas ao definir a cadeia de caracteres para representação externa do tipo, deve ser lembrado que pode ser necessário escrever um analisador completo e robusto para esta representação como função de entrada. Por exemplo:

```
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
    char      *str = PG_GETARG_CSTRING(0);
    double      x,
                y;
    Complex    *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errmsg("sintaxe de entrada inválida para complex: \"%s\"",
                        str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

A função de saída pode ser simplesmente:

```
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
```

```

{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    char        *result;

    result = (char *) palloc(100);
    snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}

```

Deve-se ter o cuidado de tornar as funções de entrada e saída inversas entre si. Se isto não for feito, vão ocorrer sérios problemas quando os dados forem salvos em um arquivo e depois lidos a partir deste arquivo. Este é um problema particularmente comum quando estão envolvidos números de ponto flutuante.

Um tipo definido pelo usuário pode possuir, opcionalmente, rotinas de entrada e saída binárias. Normalmente a entrada e saída binárias são mais rápidas, mas menos portáteis que a entrada e saída na forma de texto. Assim como a entrada e saída na forma de texto, a definição exata da representação binária fica a cargo do desenvolvedor. A maioria dos tipos de dado nativos tentam fornecer uma representação binária independente de máquina. Para o tipo `complex` será tirado proveito dos conversores para entrada e saída binária do tipo `float8`:

```

PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}

```

Para definir o tipo `complex` é necessário criar as funções de entrada e saída definidas pelo usuário antes de criar o tipo:

```

CREATE FUNCTION complex_in(cstring)
RETURNS complex
AS 'nome_do_arquivo'
LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
RETURNS cstring
AS 'nome_do_arquivo'
LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
RETURNS complex
AS 'nome_do_arquivo'
LANGUAGE C IMMUTABLE STRICT;

```

```
CREATE FUNCTION complex_send(complex)
    RETURNS bytea
    AS 'nome_do_arquivo'
    LANGUAGE C IMMUTABLE STRICT;
```

Deve ser observado que as declarações das funções de entrada e de saída fazem referência a um tipo ainda não definido. Isto é permitido, mas causa mensagens de advertência que podem ser ignoradas. A função de entrada deve vir primeiro.

Por fim o tipo de dado pode ser declarado:

```
CREATE TYPE complex (
    internallength = 16,
    input = complex_in,
    output = complex_out,
    receive = complex_recv,
    send = complex_send,
    alignment = double
);
```

Ao se definir um novo tipo base, o PostgreSQL disponibiliza automaticamente suporte para matrizes deste tipo. Por motivos históricos o tipo da matriz possui o mesmo nome do tipo base, com o caractere sublinhado (\_) prefixado.

Uma vez que o tipo de dado tenha passado a existir, podem ser declaradas funções adicionais para disponibilizar operações úteis para o tipo de dado. Em seguida podem ser definidos operadores por cima das funções e, se houver necessidade, podem ser criadas classes de operadores para dar suporte à indexação deste tipo de dado. Estas camadas adicionais são vistas nas próximas seções.

Se os valores do tipo de dado puderem exceder algumas poucas centenas de bytes em tamanho (na forma interna), o tipo de dado deve ser tornado fatiável (TOAST-able) (consulte a Seção 49.2). Para se fazer isto, a representação interna deve seguir a organização padrão para dados de comprimento variável: os primeiros quatro bytes devem ser um `int32` contendo o comprimento total, em bytes, do datum (incluindo a si próprio). As funções C que operam no tipo de dado devem tomar o cuidado de desempacotar os valores fatiados manuseados utilizando `PG_DETOAST_DATUM` (Este detalhe é geralmente escondido definindo-se macros `GETARG` específicas para o tipo). Depois, ao executar o comando `CREATE TYPE`, o comprimento interno deve ser especificado como `variable`, e selecionada a opção de armazenamento apropriada.

Para obter mais detalhes deve ser consultada a descrição do comando `CREATE TYPE`.

## 31.12. Operadores definidos pelo usuário

Todo operador é um “adoçamento sintático” para chamada a uma função subjacente que realiza o trabalho real; portanto, primeiro deve ser criada a função subjacente para depois ser criado o operador. Entretanto o operador *não é meramente* um adoçamento sintático, porque possui informações adicionais para ajudar o otimizador de comandos a otimizar os comandos que utilizam o operador. A próxima seção se dedica a explicar estas informações adicionais.

O PostgreSQL dá suporte a operador unário esquerdo, unário direito e binário. Os operadores podem ser sobrecarregados, ou seja, o mesmo nome de operador pode ser utilizado por operadores diferentes que possuam número ou tipo diferente de operandos. Quando o comando é executado, o sistema determina o operador a ser chamado a partir do número e tipo dos operandos fornecidos.

Abaixo segue um exemplo da criação de um operador para adicionar dois números complexos. É assumido que já se tenha criado a definição do tipo `complex` (consulte a Seção 31.11). Primeiro é necessária uma função para fazer o trabalho, para depois ser definido o operador:

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'nome_do_arquivo', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
```

```

    commutator = +
);

```

Agora é possível executar um comando como este:

```

SELECT (a + b) AS c FROM test_complex;

      c
-----
( 5.2, 6.05 )
(133.42, 144.95 )

```

Aqui foi mostrado como criar um operador binário. Para criar um operador binário deve-se apenas omitir `leftarg` (para operadores unários esquerdo) ou `rightarg` (para operadores unários direito). A cláusula `procedure` e as cláusulas de argumento são os únicos itens requeridos pelo comando `CREATE OPERATOR`. A cláusula `commutator` mostrada no exemplo é uma dica opcional para o otimizador de comandos. Na próxima seção podem ser obtidos mais detalhes sobre a cláusula `commutator` e outras dicas para o otimizador.

### 31.13. Informações de otimização do operador

No PostgreSQL a definição do operador pode incluir diversas cláusulas opcionais contendo informações sobre como o operador se comporta, úteis ao sistema. Estas cláusulas devem ser fornecidas sempre que for apropriado, porque podem acelerar muito os comandos que utilizam o operador. Mas se forem fornecidas, é preciso haver certeza que estão corretas! A utilização incorreta de uma cláusula de otimização pode resultar na queda do servidor, uma saída sutilmente errada, e outras coisas ruins. A cláusula de otimização sempre pode ser deixada de fora quando não há certeza sobre a mesma; a única consequência pode ser o comando demorar mais tempo para executar que o necessário.

Poderão ser adicionadas cláusulas de otimização nas versões futuras do PostgreSQL. As cláusulas aqui descritas são as que a versão 8.0.0 compreende.

#### 31.13.1. COMMUTATOR

Se for fornecida a cláusula `COMMUTATOR`, esta informa o nome do operador que é o operador de comutação do operador sendo definido. Se diz que o operador A é o operador de comutação do operador B, se  $(x \text{ A } y)$  for igual a  $(y \text{ B } x)$ , para todas as entradas possíveis  $x, y$ . Deve ser observado que B também é o operador de comutação de A. Por exemplo, para um determinado tipo de dado os operadores `<` e `>` geralmente são o operador de comutação um do outro, e o operador `+` geralmente é o operador de comutação dele mesmo. Porém, o operador `-` geralmente não é o operador de comutação de nenhum outro.

O tipo do operando esquerdo de um operador comutável é idêntico ao tipo do operando direito de seu operador de comutação, e vice-versa. Portanto, o nome do operador de comutação é tudo que precisa ser informado ao PostgreSQL para que este procure pelo operador de comutação, e é tudo que precisa ser informado na cláusula `COMMUTATOR`.

É crítico fornecer a informação sobre o operador de comutação para os operadores a serem utilizados em cláusulas de índice e de junção, porque isto permite ao otimizador de comandos “girar” a cláusula para que esta atenda a uma das formas requeridas por algum dos diferentes tipos de plano. Por exemplo, considere uma consulta com uma cláusula `WHERE` do tipo `tab1.x = tab2.y`, onde `tab1.x` e `tab2.y` são de um tipo definido pelo usuário, e suponha que `tab2.y` seja indexada. O otimizador não poderá gerar uma varredura de índice a menos que possa determinar como girar a cláusula para que se torne `tab2.y = tab1.x`, porque a maquinaria de varredura de índice espera encontrar a coluna indexada à esquerda do operador fornecido. O PostgreSQL *não* vai simplesmente assumir que esta é uma transformação válida — quem cria o operador `=` deve especificar que isto é válido, marcando o operador com a informação sobre o operador de comutação.

Quando se está definindo um operador autocomutativo é simples e direto. Agora, quando se está definindo um par de operadores de comutação a situação fica mais complicada: como é possível o primeiro operador definido fazer referência ao segundo que ainda não foi definido? Existem duas soluções para este problema:

- Uma forma é omitir a cláusula `COMMUTATOR` no primeiro operador definido, e depois especificar na definição do segundo operador. Uma vez que o PostgreSQL sabe que os operadores de comutação aparecem aos pares, quando encontra a segunda definição retorna automaticamente e preenche a cláusula `COMMUTATOR` que ficou faltando na primeira definição.

- A outra forma, mais direta, é simplesmente incluir as cláusulas `COMMUTATOR` nas duas definições. O PostgreSQL processa a primeira definição e percebe que a cláusula `COMMUTATOR` faz referência a um operador que não existe. O sistema então cria uma entrada fictícia para este operador no catálogo do sistema. Esta entrada fictícia possui entrada válida apenas para o nome do operador, os tipos de dado dos operandos da esquerda e da direita, e o tipo de dado do resultado, uma vez que isto é tudo que o PostgreSQL pode deduzir neste instante. A entrada no catálogo deste operador vai ficar vinculada a esta entrada fictícia. Posteriormente, quando o segundo operador for definido, o sistema atualiza a entrada fictícia com as informações adicionais obtidas a partir da segunda definição. No caso de se tentar utilizar o operador fictício antes deste ser preenchido, será recebida uma mensagem de erro.

### 31.13.2. NEGATOR

A cláusula `NEGATOR`, se estiver presente, declara o nome de um operador que é o operador de negação do operador sendo definido. Se diz que o operador `A` é o operador de negação do operador `B` se ambos retornam resultados booleanos, e  $(x \text{ A } y)$  é igual a `NÃO`  $(x \text{ B } y)$  para todas as entradas possíveis de `x`, `y`. Deve ser observado que `B` também é o operador de negação de `A`. Por exemplo, `< e >=` é um par de operadores de negação para a maior parte dos tipos de dado. Nunca é válido um operador ser seu próprio operador de negação.

Ao contrário dos operadores de comutação, pode ser válido existir um par de operadores unários marcados como operadores de negação um do outro; isto significa que  $(A \text{ x})$  é igual a `NÃO`  $(B \text{ x})$  para todo `x`, ou o equivalente para os operadores unários direito.

O operador de negação do operador deve possuir o mesmo tipo de dado do operando esquerdo e/ou direito do operador sendo definido. Portanto, da mesma forma que em `COMMUTATOR`, somente é necessário especificar o nome do operador na cláusula `NEGATOR`.

Fornecer o operador de negação é muito útil para o otimizador de comandos, por permitir que expressões como `NOT (x = y)` sejam simplificadas como `x <> y`. Isto ocorre com mais frequência do que se imagina, porque podem ser inseridas operações `NOT` como consequência de outras rearrumações.

Os pares de operadores de negação podem ser definidos utilizando os mesmos métodos explicados acima para os pares de operadores de comutação.

### 31.13.3. RESTRICT

A cláusula `RESTRICT`, se estiver presente, declara o nome de uma função estimadora de seletividade de restrição para o operador (Deve ser observado que é declarado o nome da função, e não o nome do operador). A cláusula `RESTRICT` só faz sentido em operador binário que retorna o tipo `boolean`. A idéia por trás do estimador de seletividade de restrição é adivinhar a fração de linhas da tabela que satisfazem a condição da cláusula `WHERE` na forma

coluna OP constante

para o operador corrente e para um determinado valor constante. Isto ajuda o otimizador dando uma noção de quantas linhas serão eliminadas pela cláusula `WHERE` que possui esta forma (Você pode estar se perguntando o que acontece quando a constante está do lado esquerdo. Bem, é para isto que o `COMMUTATOR` existe...).

A escrita de funções estimadoras de seletividade de restrição está muito acima do escopo deste capítulo, mas felizmente geralmente é possível utilizar uma das funções estimadoras padrão do sistema em operadores definidos pelo usuário. Estas são as funções estimadoras de restrição padrão:

```
eqsel para =
neqsel para <>
scalarttsel para < ou <=
scalargtsel para > ou >=
```

Pode parecer um pouco estranho que estas sejam as categorias, mas faz sentido quando se pensa sobre isto. Geralmente = aceita apenas uma pequena fração das linhas da tabela; Geralmente <> rejeita somente uma pequena fração. < aceita uma fração que depende da constante fornecida estar na faixa de valores da coluna da tabela (que, caso seja verdade, a informação coletada pelo comando `ANALYZE` será utilizada pela função estimadora de seletividade). <= aceita uma fração um pouco maior que < para a comparação com a mesma constante, mas as frações são próximas o suficiente para não valer a pena fazer distinção entre as duas, principalmente porque não se está fazendo nada melhor que uma estimativa grosseira. Comentários semelhantes se aplicam a > e >=.

Freqüentemente pode-se utilizar `eqsel` ou `neqsel` para o caso de operadores que possuam uma seletividade muito alta ou muito baixa, mesmo que não sejam realmente uma igualdade ou desigualdade. Por exemplo, os operadores geométricos de igualdade aproximada usam `eqsel` assumindo que irão corresponder apenas a uma pequena fração das entradas na tabela.

Podem ser utilizados `scalarltsel` e `scalargtsel` para fazer comparação em tipos de dado onde a conversão em escalares numéricos para fazer comparação de intervalo faça sentido. Se for possível, o tipo de dado deve ser adicionado aos compreendidos pela função `convert_to_scalar()` no arquivo `src/backend/utils/adt/selfuncs.c` (Um dia esta função será substituída por funções por-tipo-de-dado identificadas através de uma coluna do catálogo do sistema `pg_type`; mas isto ainda não foi feito). Se não for utilizado os comandos ainda assim vão funcionar, mas as estimativas do otimizador não serão tão boas quanto poderiam ser.

Existem funções estimadoras de seletividade adicionais projetadas para operadores geométricos no arquivo `src/backend/utils/adt/geo_selfuncs.c`: `areasel`, `positionsel`, e `contsel`. Quando este texto foi escrito estas funções eram apenas stubs, mas podem ser utilizadas assim mesmo (ou melhor ainda, melhoradas).

### 31.13.4. JOIN

A cláusula `JOIN`, se estiver presente, declara o nome de uma função estimadora de seletividade de junção para o operador (Deve ser observado que é declarado o nome de uma função, e não o nome de operador). A cláusula `JOIN` só faz sentido em operador binário que retorna o tipo `boolean`. A idéia por trás do estimador de seletividade de junção é adivinhar a fração de linhas de um par de tabelas que satisfazem a condição da cláusula `WHERE` na forma

```
tabela1.coluna1 OP tabela2.coluna2
```

para o operador corrente. Assim como na cláusula `RESTRICT`, ajuda o otimizador permitindo que descubra entre várias seqüências de junção possíveis qual a que deve dar menos trabalho para ser realizada.

Da mesma maneira que antes, este capítulo não tenta explicar como escrever uma função estimadora de seletividade de junção, e apenas sugere que seja utilizada uma das funções estimadoras existentes, caso uma delas se aplique:

```
eqjoinselect para =
neqjoinselect para <>
scalarltjoinselect para < ou <=
scalargtjoinselect para > ou >=
areajoinselect para comparações baseadas em área 2D
positionjoinselect para comparações baseadas em posição 2D
contjoinselect para operações baseadas em contém 2D
```

### 31.13.5. HASHES

A cláusula `HASHES`, se estiver presente, informa ao sistema que é permitido utilizar o método de junção por hash em uma junção baseada neste operador. A cláusula `HASHES` só faz sentido em operador binário que retorna o tipo `boolean` e, na prática, o operador deve representar igualdade para algum tipo de dado.

A suposição subjacente à junção por hash é que o operador de junção só retorna verdade para pares de valores à esquerda e à direita que resultam no mesmo código de hash. Se dois valores forem colocados em receptáculos de hash diferentes, a junção nunca vai compará-los, assumindo implicitamente que o resultado do operador de junção é falso. Portanto, nunca faz sentido especificar `HASHES` para operadores que não representam igualdade.

Para ser marcado como `HASHES` o operador de junção deve estar presente em uma classe de operadores de índice hash. Isto não é exigido quando se cria o operador, uma vez que a classe de operadores que faz referência não pode existir ainda. Mas a tentativa de utilizar o operador em junções hash falham em tempo de execução quando a classe de operadores não existe. O sistema precisa da classe de operadores para encontrar a função de hash específica do tipo de dado, para o tipo de dado de entrada do operador. Obviamente, antes de ser possível criar a classe de operadores é necessário criar uma função de hash adequada.

Deve ser tomado cuidado ao preparar a função de hash, porque existem formas dependentes de máquina pelas quais a função pode deixar de funcionar corretamente. Por exemplo, quando o tipo de dado é uma estrutura onde existem bits de preenchimento que não interessam, não se pode simplesmente passar toda a estrutura para a função `hash_any` (A menos que se escreva outros operadores e funções para garantir que os bits não utilizados sejam sempre zero, que é a estratégia recomendada). Outro exemplo são as máquinas que seguem o padrão IEEE para valores de ponto flutuante. Nestas máquinas zero negativo e zero positivo são valores diferentes (padrões de bit diferentes), mas definidos como sendo iguais



na comparação. Se o valor de ponto flutuante puder conter zero negativo, então são necessários passos adicionais para garantir que este gera o mesmo valor de `hash` que o zero positivo.

**Nota:** A função subjacente ao operador juntável por `hash` deve ser marcada como imutável ou estável. Se for volátil, o sistema nunca vai tentar utilizar o operador para uma junção por `hash`.

**Nota:** Se o operador juntável por `hash` possuir uma função subjacente marcada como estrita, a função também deve ser completa, ou seja, a função deve retornar verdade ou falso, e nunca nulo, para quaisquer duas entradas não nulas. Se esta regra não for seguida, a otimização de `hash` nas operações `IN` podem gerar resultados errados (Especificamente, `IN` deve retornar falso onde a resposta correta de acordo com o padrão seria nulo; ou pode gerar um erro reclamando que não foi preparada para resultado nulo).

### 31.13.6. MERGES (SORT1, SORT2, LTCMP, GTCMP)

A cláusula `MERGES`, se estiver presente, informa ao sistema que é permitido utilizar o método de junção por mesclagem (`merge`) em uma junção baseada neste operador. A cláusula `MERGES` só faz sentido em operador binário que retorna o tipo `boolean` e, na prática, o operador deve representar igualdade para algum tipo de dado ou par de tipos de dado.

A junção por mesclagem se baseia na idéia de ordenar as tabelas da esquerda e da direita primeiro, para depois varre-las em paralelo. Portanto, os dois tipos de dado devem ser capaz de ser totalmente ordenados, e o operador de junção deve ser um que somente seja bem sucedido para pares de valores que “caiam no mesmo lugar” na ordem de classificação. Na prática isto significa que o operador de junção deve se comportar como igualdade, mas diferentemente da junção por `hash`, onde devem ser o mesmo (ou pelo menos equivalente bit a bit), é possível fazer a junção por mesclagem de dois tipos de dado distintos, desde que sejam binariamente compatíveis. Por exemplo, o operador de igualdade `smallint-versus-integer` é juntável por mesclagem. Somente se necessita de operadores de classificação que coloquem os dois tipos de dado em uma sequência logicamente compatível.

A execução de uma junção por mesclagem requer que o sistema seja capaz de identificar quatro operadores relacionados com o operador de junção por mesclagem: comparação menor-que para o tipo de dado do operando à esquerda, comparação menor-que para o tipo de dado do operando à direita, comparação menor-que entre os dois tipos de dado, e comparação maior-que entre os dois tipos de dado (Na verdade são quatro operadores distintos quando o operador juntável por mesclagem possui tipos de dado dos operandos diferentes, mas quando os tipos de dado dos operandos são o mesmo os três operadores menor-que são o mesmo). É possível especificar estes operadores individualmente por nome, pelas opções `SORT1`, `SORT2`, `LTCMP` e `GTCMP`, respectivamente. O sistema preenche os nomes padrão `<`, `<`, `<`, `>`, respectivamente, quando um destes é omitido ao se especificar `MERGES`. Também, `MERGES` é assumido como implicado quando alguma destas quatro opções de operador aparece, portanto é possível especificar apenas algumas delas e deixar o sistema preencher o restante.

Os tipos de dado dos operandos dos quatro operadores de comparação podem ser deduzidos a partir dos tipos dos operandos do operador juntável por mesclagem e, portanto, da mesma maneira que em `COMMUTATOR`, somente é necessário fornecer o nome do operador nestas cláusulas. A menos que se esteja utilizando escolhas peculiares para os nomes dos operadores, basta escrever `MERGES` e deixar o sistema preencher os detalhes (Da mesma forma que em `COMMUTATOR` e `NEGATOR`, o sistema tem condição de criar entradas de operador fictícias se for definido o operador de igualdade antes dos demais).

Existem restrições adicionais para os operadores marcados como juntáveis por mesclagem. No momento estas restrições não são verificadas pelo comando `CREATE OPERATOR`, mas podem ocorrer erros ao se utilizar o operador quando alguma destas restrições não é atendida:

- Um operador de igualdade juntável por mesclagem deve possuir um comutador juntável por mesclagem (ele mesmo, se os tipos de dado dos operandos forem o mesmo, ou um operador de igualdade relacionado se forem diferentes).
- Caso exista um operador juntável por mesclagem relacionando quaisquer dois tipos de dado A e B, e outro operador juntável por mesclagem relacionando B com um terceiro tipo de dado C, então A e C também devem ter um operador juntável por mesclagem; em outras palavras, possuir um operador juntável por mesclagem deve ser transitivo.
- Podem acontecer resultados estranhos em tempo de execução quando os quatro operadores de comparação declarados não classificarem os dados de forma compatível.

**Nota:** A junção subjacente ao operador juntável por mesclagem deve ser marcada como imutável ou estável. Se for volátil, o sistema nunca tentará utilizar o operador em uma junção por mesclagem.

**Nota:** Nas versões do PostgreSQL anteriores a 7.3, `MERGES` não estava disponível: para construir um operador juntável por mesclagem era necessário escrever `SORT1` e `SORT2` explicitamente. E, também, as opções `LTCMP` e `GTCMP` não existiam; os nomes destes operadores eram estabelecidos como `< e >`, respectivamente.

## 31.14. Interfacing Extensions To Indexes

The procedures described thus far let you define new types, new functions, and new operators. However, we cannot yet define an index on a column of a new data type. To do this, we must define an *operator class* for the new data type. Later in this section, we will illustrate this concept in an example: a new operator class for the B-tree index method that stores and sorts complex numbers in ascending absolute value order.

**Nota:** Prior to PostgreSQL release 7.3, it was necessary to make manual additions to the system catalogs `pg_amop`, `pg_amproc`, and `pg_opclass` in order to create a user-defined operator class. That approach is now deprecated in favor of using `CREATE OPERATOR CLASS`, which is a much simpler and less error-prone way of creating the necessary catalog entries.

### 31.14.1. Index Methods and Operator Classes

The `pg_am` table contains one row for every index method (internally known as access method). Support for regular access to tables is built into PostgreSQL, but all index methods are described in `pg_am`. It is possible to add a new index method by defining the required interface routines and then creating a row in `pg_am` — but that is far beyond the scope of this chapter.

The routines for an index method do not directly know anything about the data types that the index method will operate on. Instead, an *operator class* identifies the set of operations that the index method needs to use to work with a particular data type. Operator classes are so called because one thing they specify is the set of `WHERE`-clause operators that can be used with an index (i.e., can be converted into an index-scan qualification). An operator class may also specify some *support procedures* that are needed by the internal operations of the index method, but do not directly correspond to any `WHERE`-clause operator that can be used with the index.

It is possible to define multiple operator classes for the same data type and index method. By doing this, multiple sets of indexing semantics can be defined for a single data type. For example, a B-tree index requires a sort ordering to be defined for each data type it works on. It might be useful for a complex-number data type to have one B-tree operator class that sorts the data by complex absolute value, another that sorts by real part, and so on. Typically, one of the operator classes will be deemed most commonly useful and will be marked as the default operator class for that data type and index method.

The same operator class name can be used for several different index methods (for example, both B-tree and hash index methods have operator classes named `int4_ops`), but each such class is an independent entity and must be defined separately.

### 31.14.2. Index Method Strategies

The operators associated with an operator class are identified by “strategy numbers”, which serve to identify the semantics of each operator within the context of its operator class. For example, B-trees impose a strict ordering on keys, lesser to greater, and so operators like “less than” and “greater than or equal to” are interesting with respect to a B-tree. Because PostgreSQL allows the user to define operators, PostgreSQL cannot look at the name of an operator (e.g., `<` or `>=`) and tell what kind of comparison it is. Instead, the index method defines a set of “strategies”, which can be thought of as generalized operators. Each operator class specifies which actual operator corresponds to each strategy for a particular data type and interpretation of the index semantics.

The B-tree index method defines five strategies, shown in Tabela 31-2.

**Tabela 31-2. B-tree Strategies**

Operation	Strategy Number
less than	1
less than or equal	2
equal	3

Operation	Strategy Number
greater than or equal	4
greater than	5

Hash indexes express only bitwise equality, and so they use only one strategy, shown in Tabela 31-3.

**Tabela 31-3. Hash Strategies**

Operation	Strategy Number
equal	1

R-tree indexes express rectangle-containment relationships. They use eight strategies, shown in Tabela 31-4.

**Tabela 31-4. R-tree Strategies**

Operation	Strategy Number
left of	1
left of or overlapping	2
overlapping	3
right of or overlapping	4
right of	5
same	6
contains	7
contained by	8

GiST indexes are even more flexible: they do not have a fixed set of strategies at all. Instead, the “consistency” support routine of each particular GiST operator class interprets the strategy numbers however it likes.

Note that all strategy operators return Boolean values. In practice, all operators defined as index method strategies must return type `boolean`, since they must appear at the top level of a `WHERE` clause to be used with an index.

By the way, the `amorderstrategy` column in `pg_am` tells whether the index method supports ordered scans. Zero means it doesn't; if it does, `amorderstrategy` is the strategy number that corresponds to the ordering operator. For example, B-tree has `amorderstrategy` = 1, which is its “less than” strategy number.

### 31.14.3. Index Method Support Routines

Strategies aren't usually enough information for the system to figure out how to use an index. In practice, the index methods require additional support routines in order to work. For example, the B-tree index method must be able to compare two keys and determine whether one is greater than, equal to, or less than the other. Similarly, the R-tree index method must be able to compute intersections, unions, and sizes of rectangles. These operations do not correspond to operators used in qualifications in SQL commands; they are administrative routines used by the index methods, internally.

Just as with strategies, the operator class identifies which specific functions should play each of these roles for a given data type and semantic interpretation. The index method defines the set of functions it needs, and the operator class identifies the correct functions to use by assigning them to the “support function numbers”.

B-trees require a single support function, shown in Tabela 31-5.

**Tabela 31-5. B-tree Support Functions**

Função	Support Number
Compare two keys and return an integer less than zero, zero, or greater than zero,	1

Função	Support Number
indicating whether the first key is less than, equal to, or greater than the second.	

Hash indexes likewise require one support function, shown in Tabela 31-6.

**Tabela 31-6. Hash Support Functions**

Função	Support Number
Compute the hash value for a key	1

R-tree indexes require three support functions, shown in Tabela 31-7.

**Tabela 31-7. R-tree Support Functions**

Função	Support Number
union	1
intersection	2
size	3

GiST indexes require seven support functions, shown in Tabela 31-8.

**Tabela 31-8. GiST Support Functions**

Função	Support Number
consistent	1
union	2
compress	3
decompress	4
penalty	5
picksplit	6
equal	7

Unlike strategy operators, support functions return whichever data type the particular index method expects, for example in the case of the comparison function for B-trees, a signed integer.

#### 31.14.4. An Example

Now that we have seen the ideas, here is the promised example of creating a new operator class. (You can find a working copy of this example in `src/tutorial/complex.c` and `src/tutorial/complex.sql` in the source distribution.) The operator class encapsulates operators that sort complex numbers in absolute value order, so we choose the name `complex_abs_ops`. First, we need a set of operators. The procedure for defining operators was discussed in Seção 31.12. For an operator class on B-trees, the operators we require are:

- absolute-value less-than (strategy 1)
- absolute-value less-than-or-equal (strategy 2)
- absolute-value equal (strategy 3)
- absolute-value greater-than-or-equal (strategy 4)
- absolute-value greater-than (strategy 5)

The least error-prone way to define a related set of comparison operators is to write the B-tree comparison support function first, and then write the other functions as one-line wrappers around the support function. This reduces the odds of getting inconsistent results for corner cases. Following this approach, we first write

```
#define Mag(c) ((c)->x*(c)->x + (c)->y*(c)->y)

static int
complex_abs_cmp_internal(Complex *a, Complex *b)
{
    double      amag = Mag(a),
               bmag = Mag(b);

    if (amag < bmag)
        return -1;
    if (amag > bmag)
        return 1;
    return 0;
}
```

Now the less-than function looks like

```
PG_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(PG_FUNCTION_ARGS)
{
    Complex      *a = (Complex *) PG_GETARG_POINTER(0);
    Complex      *b = (Complex *) PG_GETARG_POINTER(1);

    PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

The other four functions differ only in how they compare the internal function's result to zero.

Next we declare the functions and the operators based on the functions to SQL:

```
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
    AS 'nome_do_arquivo', 'complex_abs_lt'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
    leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
    commutator = > , negator = >= ,
    restrict = scalarltsel, join = scalarltjoinsel
);
```

It is important to specify the correct commutator and negator operators, as well as suitable restriction and join selectivity functions, otherwise the optimizer will be unable to make effective use of the index. Note that the less-than, equal, and greater-than cases should use different selectivity functions.

Other things worth noting are happening here:

- There can only be one operator named, say, = and taking type `complex` for both operands. In this case we don't have any other operator = for `complex`, but if we were building a practical data type we'd probably want = to be the ordinary equality operation for complex numbers (and not the equality of the absolute values). In that case, we'd need to use some other operator name for `complex_abs_eq`.
- Although PostgreSQL can cope with functions having the same name as long as they have different argument data types, C can only cope with one global function having a given name. So we shouldn't name the C function something simple like `abs_eq`. Usually it's a good practice to include the data type name in the C function name, so as not to conflict with functions for other data types.
- We could have made the PostgreSQL name of the function `abs_eq`, relying on PostgreSQL to distinguish it by argument data types from any other PostgreSQL function of the same name. To keep the example simple, we make the function have the same names at the C level and PostgreSQL level.

The next step is the registration of the support routine required by B-trees. The example C code that implements this is in the same file that contains the operator functions. This is how we declare the function:

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'nome_do_arquivo'
    LANGUAGE C IMMUTABLE STRICT;
```

Now that we have the required operators and support routine, we can finally create the operator class:

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR          1          < ,
        OPERATOR          2          <= ,
        OPERATOR          3          = ,
        OPERATOR          4          >= ,
        OPERATOR          5          > ,
        FUNCTION           1          complex_abs_cmp(complex, complex);
```

And we're done! It should now be possible to create and use B-tree indexes on `complex` columns.

We could have written the operator entries more verbosely, as in

```
OPERATOR          1          < (complex, complex) ,
```

but there is no need to do so when the operators take the same data type we are defining the operator class for.

The above example assumes that you want to make this new operator class the default B-tree operator class for the `complex` data type. If you don't, just leave out the word `DEFAULT`.

### 31.14.5. Cross-Data-Type Operator Classes

So far we have implicitly assumed that an operator class deals with only one data type. While there certainly can be only one data type in a particular index column, it is often useful to index operations that compare an indexed column to a value of a different data type. This is presently supported by the B-tree and GiST index methods.

B-trees require the left-hand operand of each operator to be the indexed data type, but the right-hand operand can be of a different type. There must be a support function having a matching signature. For example, the built-in operator class for type `bigint` (`int8`) allows cross-type comparisons to `int4` and `int2`. It could be duplicated by this definition:

```
CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree AS
    -- standard int8 comparisons
    OPERATOR 1 < ,
    OPERATOR 2 <= ,
    OPERATOR 3 = ,
    OPERATOR 4 >= ,
    OPERATOR 5 > ,
    FUNCTION 1 btint8cmp(int8, int8) ,

    -- cross-type comparisons to int2 (smallint)
    OPERATOR 1 < (int8, int2) ,
    OPERATOR 2 <= (int8, int2) ,
    OPERATOR 3 = (int8, int2) ,
    OPERATOR 4 >= (int8, int2) ,
    OPERATOR 5 > (int8, int2) ,
    FUNCTION 1 btint82cmp(int8, int2) ,

    -- cross-type comparisons to int4 (integer)
    OPERATOR 1 < (int8, int4) ,
    OPERATOR 2 <= (int8, int4) ,
    OPERATOR 3 = (int8, int4) ,
    OPERATOR 4 >= (int8, int4) ,
    OPERATOR 5 > (int8, int4) ,
```

```
FUNCTION 1 btint84cmp(int8, int4) ;
```

Notice that this definition “overloads” the operator strategy and support function numbers. This is allowed (for B-tree operator classes only) so long as each instance of a particular number has a different right-hand data type. The instances that are not cross-type are the default or primary operators of the operator class.

GiST indexes do not allow overloading of strategy or support function numbers, but it is still possible to get the effect of supporting multiple right-hand data types, by assigning a distinct strategy number to each operator that needs to be supported. The `consistent` support function must determine what it needs to do based on the strategy number, and must be prepared to accept comparison values of the appropriate data types.

### 31.14.6. System Dependencies on Operator Classes

PostgreSQL uses operator classes to infer the properties of operators in more ways than just whether they can be used with indexes. Therefore, you might want to create operator classes even if you have no intention of indexing any columns of your data type.

In particular, there are SQL features such as `ORDER BY` and `DISTINCT` that require comparison and sorting of values. To implement these features on a user-defined data type, PostgreSQL looks for the default B-tree operator class for the data type. The “equals” member of this operator class defines the system’s notion of equality of values for `GROUP BY` and `DISTINCT`, and the sort ordering imposed by the operator class defines the default `ORDER BY` ordering.

Comparison of arrays of user-defined types also relies on the semantics defined by the default B-tree operator class.

If there is no default B-tree operator class for a data type, the system will look for a default hash operator class. But since that kind of operator class only provides equality, in practice it is only enough to support array equality.

When there is no default operator class for a data type, you will get errors like “could not identify an ordering operator” if you try to use these SQL features with the data type.

**Nota:** In PostgreSQL versions before 7.4, sorting and grouping operations would implicitly use operators named `=`, `<`, and `>`. The new behavior of relying on default operator classes avoids having to make any assumption about the behavior of operators with particular names.

### 31.14.7. Special Features of Operator Classes

There are two special features of operator classes that we have not discussed yet, mainly because they are not useful with the most commonly used index methods.

Normally, declaring an operator as a member of an operator class means that the index method can retrieve exactly the set of rows that satisfy a `WHERE` condition using the operator. For example,

```
SELECT * FROM table WHERE integer_column < 4;
```

can be satisfied exactly by a B-tree index on the integer column. But there are cases where an index is useful as an inexact guide to the matching rows. For example, if an R-tree index stores only bounding boxes for objects, then it cannot exactly satisfy a `WHERE` condition that tests overlap between nonrectangular objects such as polygons. Yet we could use the index to find objects whose bounding box overlaps the bounding box of the target object, and then do the exact overlap test only on the objects found by the index. If this scenario applies, the index is said to be “lossy” for the operator, and we add `RECHECK` to the `OPERATOR` clause in the `CREATE OPERATOR CLASS` command. `RECHECK` is valid if the index is guaranteed to return all the required rows, plus perhaps some additional rows, which can be eliminated by performing the original operator invocation.

Consider again the situation where we are storing in the index only the bounding box of a complex object such as a polygon. In this case there’s not much value in storing the whole polygon in the index entry — we may as well store just a simpler object of type `box`. This situation is expressed by the `STORAGE` option in `CREATE OPERATOR CLASS`: we’d write something like

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
    ...
    STORAGE box;
```

At present, only the GiST index method supports a `STORAGE` type that's different from the column data type. The GiST `compress` and `decompress` support routines must deal with data-type conversion when `STORAGE` is used.

## Notas

1. `abstract data type` (ADT) — Um tipo de abstração de dado onde a forma interna do tipo fica escondida atrás de um conjunto de funções de acesso. Os valores do tipo são criados e inspecionados apenas pelas chamadas às funções de acesso, permitindo que a implementação do tipo seja modificada sem ser necessária qualquer modificação fora do módulo onde está definida. Os objetos e as ADTs são formas de abstração de dados, mas os objetos não são ADTs. Os objetos utilizam abstração procedural (métodos), e não abstração de tipo. Um exemplo clássico de ADT é o tipo de dado pilha, para o qual devem ser fornecidas funções para criar uma pilha vazia, para colocar elementos na pilha e para tirar elementos da pilha. FOLDOC - Free On-Line Dictionary of Computing (<http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=adt>) (N. do T.)
2. `pg_config --pkglibdir` retorna `/usr/local/pgsql/lib` na plataforma utilizada. (N. do T.)
3. `pg_config --includedir-server` retorna `/usr/local/pgsql/include/server` na plataforma utilizada. (N. do T.)
4. Para gerar o arquivo `complex.sql` primeiro este diretório deve ser tornado o diretório corrente, e depois executado `make`, conforme descrito no arquivo `README`. (N. do T.)



## Capítulo 32. Gatilhos

Este capítulo descreve como escrever funções de gatilho. As funções de gatilho podem ser escritas na linguagem C, ou em uma das várias linguagens procedurais disponíveis. No momento não é possível escrever funções de gatilho na linguagem SQL.

### 32.1. Visão geral do comportamento dos gatilhos

O gatilho pode ser definido para executar antes ou depois de uma operação de `INSERT`, `UPDATE` ou `DELETE`, tanto uma vez para cada linha modificada quanto uma vez por instrução SQL. Quando ocorre o evento do gatilho, a função de gatilho é chamada no momento apropriado para tratar o evento.

A função de gatilho deve ser definida antes do gatilho ser criado. A função de gatilho deve ser declarada como uma função que não recebe argumentos e que retorna o tipo `trigger` (A função de gatilho recebe sua entrada através de estruturas `TriggerData` passadas especialmente para estas funções, e não na forma comum de argumentos de função).

Tendo sido criada a função de gatilho adequada, o gatilho é estabelecido através do comando `CREATE TRIGGER`. A mesma função de gatilho pode ser utilizada por vários gatilhos.

Existem dois tipos de gatilhos: gatilhos-por-linha e gatilhos-por-instrução. Em um gatilho-por-linha, a função é chamada uma vez para cada linha afetada pela instrução que disparou o gatilho. Em contraste, um gatilho-por-instrução é chamado somente uma vez quando a instrução apropriada é executada, a despeito do número de linhas afetadas pela instrução. Em particular, uma instrução que não afeta nenhuma linha ainda assim resulta na execução dos gatilhos-por-instrução aplicáveis. Estes dois tipos de gatilho são algumas vezes chamados de “gatilhos no nível-de-linha” e “gatilhos no nível-de-instrução”, respectivamente.

Os gatilhos no nível-de-instrução “BEFORE” (antes) naturalmente disparam antes da instrução começar a fazer alguma coisa, enquanto os gatilhos no nível-de-instrução “AFTER” (após) disparam bem no final da instrução. Os gatilhos no nível-de-linha “BEFORE” (antes) disparam logo antes da operação em uma determinada linha, enquanto os gatilhos no nível-de-linha “AFTER” (após) disparam no fim da instrução (mas antes dos gatilhos no nível-de-instrução “AFTER”).

As funções de gatilho chamadas por gatilhos-por-instrução devem sempre retornar `NULL`. As funções de gatilho chamadas por gatilhos-por-linha podem retornar uma linha da tabela (um valor do tipo `HeapTuple`) para o executor da chamada, se assim o decidirem. Os gatilhos no nível-de-linha disparados antes de uma operação possuem as seguintes escolhas:

- Podem retornar `NULL` para saltar a operação para a linha corrente. Isto instrui ao executor a não realizar a operação no nível-de-linha que chamou o gatilho (a inserção ou a modificação de uma determinada linha da tabela).
- Para os gatilhos de `INSERT` e `UPDATE`, no nível-de-linha apenas, a linha retornada se torna a linha que será inserida ou que substituirá a linha sendo atualizada. Isto permite à função de gatilho modificar a linha sendo inserida ou atualizada.

Um gatilho no nível-de-linha, que não pretenda causar nenhum destes comportamentos, deve ter o cuidado de retornar como resultado a mesma linha que recebeu (ou seja, a linha `NEW` para os gatilhos de `INSERT` e `UPDATE`, e a linha `OLD` para os gatilhos de `DELETE`).

O valor retornado é ignorado nos gatilhos no nível-de-linha disparados após a operação e, portanto, podem muito bem retornar `NULL`.

Se for definido mais de um gatilho para o mesmo evento na mesma relação, os gatilhos são disparados pela ordem alfabética de seus nomes. No caso dos gatilhos para antes, a linha possivelmente modificada retornada por cada gatilho se torna a entrada do próximo gatilho. Se algum dos gatilhos para antes retornar `NULL`, a operação é abandonada e os gatilhos seguintes não são disparados.

Tipicamente, os gatilhos no nível-de-linha que disparam antes são utilizados para verificar ou modificar os dados que serão inseridos ou atualizados. Por exemplo, um gatilho que dispara antes pode ser utilizado para inserir a hora corrente em uma coluna do tipo `timestamp`, ou para verificar se dois elementos da linha são consistentes. Os gatilhos no nível-de-linha que disparam depois fazem mais sentido para propagar as atualizações para outras tabelas, ou fazer verificação de consistência com relação a outras tabelas. O motivo desta divisão de trabalho é porque um gatilho que dispara depois pode ter certeza de estar vendo o valor final da linha, enquanto um gatilho que dispara antes não pode ter esta certeza; podem haver outros gatilhos que disparam antes disparando após o mesmo. Se não houver nenhum motivo específico para fazer um gatilho

disparar antes ou depois, o gatilho para antes é mais eficiente, uma vez que a informação sobre a operação não precisa ser salva até o fim da instrução.

Se a função de gatilho executar comandos SQL, então estes comandos podem disparar gatilhos novamente. Isto é conhecido como cascatear gatilhos. Não existe limitação direta do número de níveis de cascateamento. É possível que o cascateamento cause chamadas recursivas do mesmo gatilho; por exemplo, um gatilho para `INSERT` pode executar um comando que insere uma linha adicional na mesma tabela, fazendo com que o gatilho para `INSERT` seja disparado novamente. É responsabilidade do programador do gatilho evitar recursões infinitas nestes casos.

Ao se definir um gatilho, podem ser especificados argumentos para o mesmo. A finalidade de se incluir argumentos na definição do gatilho é permitir que gatilhos diferentes com requisitos semelhantes chamem a mesma função. Por exemplo, pode existir uma função de gatilho generalizada que recebe como argumentos dois nomes de colunas e coloca o usuário corrente em uma e a data corrente em outra. Escrita de maneira apropriada, esta função de gatilho se torna independente da tabela específica para a qual está sendo utilizada. Portanto, a mesma função pode ser utilizada para eventos de `INSERT` em qualquer tabela com colunas apropriadas, para acompanhar automaticamente a criação de registros na tabela de transação, por exemplo. Também pode ser utilizada para acompanhar os eventos de última atualização, se for definida em um gatilho de `UPDATE`.

Cada linguagem de programação que suporta gatilhos possui o seu próprio método para tornar os dados de entrada do gatilho disponíveis para a função de gatilho. Estes dados de entrada incluem o tipo de evento do gatilho (ou seja, `INSERT` ou `UPDATE`), assim como os argumentos listados em `CREATE TRIGGER`. Para um gatilho no nível-de-linha, os dados de entrada também incluem as linhas `NEW` para os gatilhos de `INSERT` e `UPDATE`, e/ou a linha `OLD` para os gatilhos de `UPDATE` e `DELETE`. Atualmente não há maneira de examinar individualmente as linhas modificadas pela instrução nos gatilhos no nível-de-instrução.

## 32.2. Visibilidade das mudanças nos dados

Se forem executados comandos SQL na função de gatilho, e estes comandos acessarem a tabela para a qual o gatilho se destina, então deve-se estar ciente das regras de visibilidade dos dados, porque estas determinam se estes comandos SQL enxergam as mudanças nos dados para os quais o gatilho foi disparado. Em resumo:

- Os gatilhos no nível-de-instrução seguem regras simples de visibilidade: nenhuma das modificações feitas pela instrução é enxergada pelos gatilhos no nível-de-instrução chamados antes da instrução, enquanto todas as modificações são enxergadas pelos gatilhos no nível-de-instrução que disparam depois da instrução.
- As modificações nos dados (inserção, atualização e exclusão) causadoras do disparo do gatilho, naturalmente *não* são enxergadas pelos comandos SQL executados em um gatilho no nível-de-linha que dispara antes, porque ainda não ocorreram.
- Entretanto, os comandos SQL executados em um gatilho no nível-de-linha para antes *enxergam* os efeitos das modificações nos dados das linhas processadas anteriormente no mesmo comando externo. Isto requer cautela, uma vez que a ordem destes eventos de modificação geralmente não é previsível; um comando SQL que afeta várias linhas pode atuar sobre as linhas em qualquer ordem.
- Quando é disparado um gatilho no nível-de-linha para depois, todas as modificações nos dados feitas pelo comando externo já estão completas, sendo enxergadas pela função de gatilho chamada.

Podem ser encontradas informações adicionais sobre as regras de visibilidade dos dados na Seção 39.4. O exemplo na Seção 32.4 contém uma demonstração destas regras.

## 32.3. Escrita de funções de gatilho em C

Esta seção descreve os detalhes de baixo nível da interface com a função de gatilho. Estas informações somente são necessárias para se escrever funções de gatilho em C. Se estiver sendo utilizada uma linguagem de nível mais alto, então estes detalhes são tratados para você. A documentação de cada linguagem procedural explica como escrever gatilhos nesta linguagem.

A função de gatilho deve utilizar a interface de gerência de função “versão 1”.

Quando uma função é chamada pelo gerenciador de gatilho não é passado nenhum argumento normal, mas é passado um ponteiro de “contexto” apontando para a estrutura `TriggerData`. As funções em C podem verificar se foram chamadas pelo gerenciador de gatilhos executando a macro

```
CALLED_AS_TRIGGER(fcinfo)
```

que expande para

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

Se retornar verdade, então é seguro converter `fcinfo->context` no tipo `TriggerData` \* e fazer uso da estrutura `TriggerData` apontada. A função *não* deve alterar a estrutura de `TriggerData` ou de qualquer dado apontado por esta.

A struct `TriggerData` está definida em `commands/trigger.h`:

```
typedef struct TriggerData
{
    NodeTag      type;
    TriggerEvent  tg_event;
    Relation      tg_relation;
    HeapTuple     tg_trigtuple;
    HeapTuple     tg_newtuple;
    Trigger       *tg_trigger;
    Buffer         tg_trigtuplebuf;
    Buffer         tg_newtuplebuf;
} TriggerData;
```

onde os membros estão definidos conforme mostrado abaixo:

type

Sempre `T_TriggerData`.

tg\_event

Descreve o evento para o qual a função foi chamada. Podem ser utilizadas as seguintes macros para examinar `tg_event`:

`TRIGGER_FIRED_BEFORE(tg_event)`

Retorna verdade se o gatilho disparou antes da operação.

`TRIGGER_FIRED_AFTER(tg_event)`

Retorna verdade se o gatilho disparou depois da operação.

`TRIGGER_FIRED_FOR_ROW(tg_event)`

Retorna verdade se o gatilho disparou para um evento no nível-de-linha.

`TRIGGER_FIRED_FOR_STATEMENT(tg_event)`

Retorna verdade se o gatilho disparou para um evento no nível-de-instrução.

`TRIGGER_FIRED_BY_INSERT(tg_event)`

Retorna verdade se o gatilho foi disparado por um comando `INSERT`.

`TRIGGER_FIRED_BY_UPDATE(tg_event)`

Retorna verdade se o gatilho foi disparado por um comando `UPDATE`.

`TRIGGER_FIRED_BY_DELETE(tg_event)`

Retorna verdade se o gatilho foi disparado por um comando `DELETE`.

tg\_relation

Um ponteiro para a estrutura que descreve a relação para a qual o gatilho foi disparado. Os detalhes sobre esta estrutura devem ser procurados no arquivo `utils/rel.h`. Os itens mais interessantes são `tg_relation->rd_att` (descriptor das tuplas da relação) e `tg_relation->rd_rel->relname` (nome da relação; o tipo não é `char*`, e sim `NameData`; deve ser utilizado `SPI_getrelname(tg_relation)` para obter `char*` se for necessário copiar o nome).

`tg_trigtuple`

Ponteiro para a linha para a qual o gatilho foi disparado. Esta é a linha sendo inserida, atualizada ou excluída. Se este gatilho foi disparado por um `INSERT` ou `DELETE`, então é o que deve ser retornado pela função se não for desejado substituir a linha por outra diferente (no caso do `INSERT`) ou saltar a operação.

`tg_newtuple`

Ponteiro para a nova versão da linha, se o gatilho foi disparado por um `UPDATE`, ou `NULL`, se foi disparado por um `INSERT` ou `DELETE`. É o que deve ser retornado pela função se o evento for um `UPDATE` e não for desejado substituir a linha por outra diferente ou saltar a operação.

`tg_trigger`

Um ponteiro para a estrutura do tipo `Trigger`, definida no arquivo `utils/rel.h`:

```
typedef struct Trigger
{
    Oid          tgoid;
    char         *tgname;
    Oid          tgfoid;
    int16        tgtype;
    bool         tgenabled;
    bool         tgisconstraint;
    Oid          tgconstrrelid;
    bool         tgdeferrable;
    bool         tginitdeferred;
    int16        tgnargs;
    int16        tgattr[FUNC_MAX_ARGS];
    char         **tgargs;
} Trigger;
```

onde `tgname` é o nome do gatilho, `tgnargs` é o número de argumentos em `tgargs`, e `tgargs` é uma matriz de ponteiros para os argumentos especificados na declaração `CREATE TRIGGER`. Os outros membros são para uso interno apenas.

`tg_trigtuplebuf`

Um buffer contendo `tg_trigtuple`, ou `InvalidBuffer` se a tupla não existir ou não estiver armazenada em um buffer de disco.

`tg_newtuplebuf`

Um buffer contendo `tg_newtuple`, ou `InvalidBuffer` se a tupla não existir ou não estiver armazenada em um buffer de disco.

Uma função de gatilho deve retornar um ponteiro para `HeapTuple` ou um ponteiro `NULL` (*não* um valor SQL nulo, ou seja, não se deve definir `isNull` como verdade). Deve-se tomar o cuidado de retornar `tg_trigtuple` ou `tg_newtuple`, conforme seja apropriado, se não for desejado modificar a linha onde está sendo realizada a operação.

## 32.4. Um exemplo completo

Abaixo está mostrado um exemplo bem simples de uma função de gatilho escrita em C (Podem ser encontrados na documentação das linguagens procedurais exemplos de gatilhos escritos nestas linguagens procedurais).

A função `trigf` informa o número de linhas na tabela `ttest`, e salta a operação se o comando tentar inserir um valor nulo na coluna `x` (Portanto, o gatilho age como uma restrição de não nulo, mas não interrompe a transação).

Primeiro, a definição da tabela:

```
CREATE TABLE ttest (
    x integer
);
```

A seguir se encontra o código fonte da função de gatilho:

```

#include "postgres.h"
#include "executor/spi.h"      /* necessário para trabalhar com SPI */
#include "commands/trigger.h" /* ... e gatilhos */

extern Datum trigf(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char        *when;
    bool        checknull = false;
    bool        isnull;
    int         ret, i;

    /* certificar-se que foi chamado como um gatilho */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: não foi chamada por um gerenciador de gatilho");

    /* tupla a ser retornada para o executor */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* verificar valores nulos */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;

    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "antes ";
    else
        when = "depois";

    tupdesc = trigdata->tg_relation->rd_att;

    /* conectar ao gerenciador de SPI */
    if ((ret = SPI_connect()) < 0)
        elog(INFO, "trigf (disparado %s): SPI_connect returned %d", when, ret);

    /* obter o número de linhas na tabela */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(NOTICE, "trigf (disparado %s): SPI_exec retornou %d", when, ret);

    /* count(*) retorna int8, deve-se ter cuidado ao converter */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                     SPI_tuptable->tupdesc,
                                     1,
                                     &isnull));

    elog(INFO, "trigf (disparado %s): existem %d linhas em ttest", when, i);

    SPI_finish();

    if (checknull)

```

```

    {
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}

```

Após compilar o código fonte, a função e o gatilho são declarados:

```

CREATE FUNCTION trigf() RETURNS trigger
    AS 'nome_do_arquivo'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();

```

Agora pode ser testada a operação do gatilho:

```

=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (disparado antes): existem 0 linhas em ttest
INSERT 0 0

-- Inserção saltada e gatilho AFTER não é disparado

=> SELECT * FROM ttest;

x
---
(0 linhas)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (disparado antes ): existem 0 linhas em ttest
INFO:  trigf (disparado depois): existem 1 linhas em ttest
          ^^^^^^^^^
          lembre-se do que foi dito sobre visibilidade.
INSERT 167793 1
vac=> SELECT * FROM ttest;

x
---
1
(1 linha)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (disparado antes ): existem 1 linhas em ttest
INFO:  trigf (disparado depois): existem 2 linhas em ttest
          ^^^^^^^^^
          lembre-se do que foi dito sobre visibilidade.
INSERT 167794 1
=> SELECT * FROM ttest;

x
---
1
2
(2 linhas)

=> UPDATE ttest SET x = NULL WHERE x = 2;

```

```

INFO:  trigf (disparado antes ): existem 2 linhas em ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (disparado antes ): existem 2 linhas em ttest
INFO:  trigf (disparado depois): existem 2 linhas em ttest
UPDATE 1
vac=> SELECT * FROM ttest;
      x
---
      1
      4
(2 linhas)
=> DELETE FROM ttest;
INFO:  trigf (disparado antes ): existem 2 linhas em ttest
INFO:  trigf (disparado depois): existem 1 linhas em ttest
INFO:  trigf (disparado antes ): existem 1 linhas em ttest
INFO:  trigf (disparado depois): existem 0 linhas em ttest
                        ^^^^^^^^

      lembre-se do que foi dito sobre visibilidade.
DELETE 2
=> SELECT * FROM ttest;
      x
---
(0 linhas)

```

Existem exemplos mais complexos no arquivo `src/test/regress/regress.c` e no diretório `contrib/spi`.

## Capítulo 33. O sistema de regras

Este capítulo discute o sistema de regras do PostgreSQL. Os sistemas de regras de produção são conceitualmente simples, mas existem vários pontos delicados envolvidos na utilização destes sistemas.

Alguns outros sistemas de banco de dados definem regras de banco de dados ativas, que geralmente são procedimentos armazenados e gatilhos. No PostgreSQL estas regras podem ser implementadas utilizando funções e gatilhos também.

O sistema de regras (falando mais precisamente, o sistema de regras de reescrita de comandos) é totalmente diferente de procedimentos armazenados e gatilhos. O sistema de regras modifica o comando para levar as regras em consideração, e depois passa o comando modificado para o planejador de comandos para planejamento e execução. É muito poderoso, e pode ser utilizado para muitas finalidades como procedimentos de linguagem de comando, visões e versões. Os fundamentos teóricos e o poder deste sistema de regras estão discutidos também em *On Rules, Procedures, Caching and Views in Database Systems* e *A Unified Framework for Version Modeling Using Production Rules in a Database System*.

### 33.1. A árvore de comando

Para entender como o sistema de regras funciona, é necessário saber quando é chamado e quais são suas entradas e resultados.

O sistema de regras fica localizado entre o analisador e o planejador. Recebe a saída do analisador, uma árvore de comando, e as regras de reescrita definidas pelo usuário, que também são árvores de comando com algumas informações adicionais, e cria zero ou mais árvores de comando como resultado. Portanto, sua entrada e saída são sempre alguma coisa que poderia ter sido produzida pelo próprio analisador e, portanto, qualquer coisa que o sistema de regras enxerga é basicamente representável como uma instrução SQL.

Agora, o que é uma árvore de comando? É a representação interna da instrução SQL, onde as partes elementares com as quais a instrução é construída são armazenadas separadamente. As árvores de comando podem ser mostradas no log do servidor, se forem definidos os parâmetros de configuração `debug_print_parse`, `debug_print_rewritten` ou `debug_print_plan`. As ações da regra também são armazenadas como árvores de comando no catálogo do sistema `pg_rewrite`. Não são formatadas como a saída do log, mas contêm exatamente a mesma informação.

Ler a árvore de comando diretamente requer alguma experiência. Mas uma vez que as representações SQL das árvores de comando são suficientes para compreender o sistema de regras, este capítulo não ensina como ler estas árvores.

Para ler as representações SQL das árvores de comando presentes neste capítulo, é necessário ser capaz de identificar as partes em que a instrução fica dividida quando está na estrutura da árvore de comando. As partes da árvore de comando são:

o tipo do comando

Este é um valor simples, informando qual comando (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) produziu a árvore de comando.

a tabela de abrangência

A tabela de abrangência é a lista de relações utilizadas no comando. Em uma instrução `SELECT` são as relações presentes após a palavra chave `FROM`.

Toda entrada na tabela de abrangência identifica uma tabela ou visão, e informa por qual nome esta é chamada nas outras partes do comando. Na árvore de comando as entradas na tabela de abrangência são referenciadas por número em vez de por nome, portanto não importa se há nomes duplicados, como seria o caso em uma instrução SQL. Isto pode acontecer após as tabelas de abrangência das regras serem mescladas. Nos exemplos deste capítulo esta situação não ocorre.

a relação do resultado

Este valor é um índice para a tabela de abrangência que identifica a relação para onde os resultados do comando vão.

Os comandos `SELECT` normalmente não possuem uma relação do resultado. O caso especial do `SELECT INTO` é praticamente idêntico ao comando `CREATE TABLE` seguido pelo comando `INSERT ... SELECT`, não sendo discutido separadamente aqui.



Para os comandos `INSERT`, `UPDATE` e `DELETE`, a relação do resultado é a tabela (ou visão!) onde as alterações vão ocorrer.

a lista de destino

a lista de destino é uma lista de expressões que definem o resultado do comando. No caso do `SELECT`, estas expressões são aquelas que constroem a saída final da consulta. Correspondem às expressões entre as palavras chave `SELECT` e `FROM` (\* é apenas uma abreviatura de todos os nomes de coluna da relação. É expandido pelo analisador em colunas individuais e, portanto, o sistema de regras nunca o vê).

Os comandos `DELETE` não necessitam de uma lista de destino, porque não produzem nenhum resultado. Na verdade, o planejador adiciona a entrada especial `CTID` à lista de destino vazia, mas isto é após o sistema de regras e será discutido posteriormente; para o sistema de regras, a lista de destino é vazia.

Para os comandos `INSERT`, a lista de destino descreve as novas linhas que devem ir para a relação do resultado. Consiste das expressões na cláusula `VALUES`, ou das que vêm da cláusula `SELECT` em `INSERT ... SELECT`. O primeiro passo do processo de reescrita adiciona entradas na lista de destino para todas as colunas que não receberam atribuições pelo comando original, mas têm valor padrão. Todas as colunas restantes (sem valor atribuído nem valor padrão) são preenchidas pelo planejador com uma expressão nula constante.

Para os comandos `UPDATE`, a lista de destino descreve as novas linhas que substituirão as antigas. No sistema de regras, contém apenas as expressões da parte `SET coluna = expressão` do comando. O planejador trata as colunas que faltam inserindo expressões que copiam os valores da linha antiga para a linha nova. Também adiciona a entrada especial `CTID`, da mesma maneira que para o `DELETE`.

Toda entrada na lista de destino contém uma expressão que pode ser um valor constante, uma variável apontando para uma coluna de uma das relações da tabela de abrangência, um parâmetro ou uma árvore de expressão feita de chamadas a função, constantes, variáveis, operadores, etc.

a qualificação

A qualificação do comando é uma expressão muito parecida com as contidas nas entradas da lista de destino. O valor do resultado desta expressão é um valor booleano que informa se a operação (`INSERT`, `UPDATE`, `DELETE` ou `SELECT`) para a linha do resultado final deve ser executada ou não. Corresponde a cláusula `WHERE` de uma instrução SQL.

a árvore de junção

A árvore de junção do comando mostra a estrutura da cláusula `FROM`. Para uma consulta simples, como `SELECT ... FROM a, b, c`, a árvore de junção é apenas uma lista de itens do `FROM`, porque é permitido fazer a junção em qualquer ordem, mas quando são utilizadas expressões `JOIN`, em particular as junções externas, é necessário fazer a junção na ordem mostrada pelas expressões `JOIN`. Neste caso, a árvore de junção mostra a estrutura das expressões `JOIN`. As restrições associadas a uma determinada cláusula `JOIN` (das expressões `ON` ou `USING`) são armazenadas como expressões de qualificação anexadas a estes nodos de árvore de junção. Também torna-se conveniente armazenar a expressão `WHERE` de nível mais alto como uma qualificação anexada ao item de nível mais alto da árvore de junção. Portanto, na realidade a árvore de junção representa as cláusulas `FROM` e `WHERE` do `SELECT`.

as outras

As outras partes da árvore de comando, como a cláusula `ORDER BY`, não são de interesse aqui. O sistema de regras substitui algumas entradas nestas partes ao aplicar as regras, mas isto não tem muito a ver com os fundamentos do sistema de regras.

## 33.2. As visões e o sistema de regras

No PostgreSQL as visões são implementadas através do sistema de regras. De fato, essencialmente não há diferença entre

```
CREATE VIEW minha_visão AS SELECT * FROM minha_tabela;
```

quando se compara com os dois comandos

```
CREATE TABLE minha_visão (mesma lista de colunas de minha_tabela);
CREATE RULE "_RETURN" AS ON SELECT TO minha_visão DO INSTEAD
    SELECT * FROM minha_tabela;
```

porque é exatamente isto o que o comando `CREATE VIEW` faz internamente. Causa alguns efeitos colaterais. Um deles é que nos catálogos do sistema do PostgreSQL as informações sobre visão são exatamente as mesmas que para tabela. Portanto, para o analisador não existe absolutamente nenhuma diferença entre uma tabela e uma visão, são a mesma coisa: relações.

### 33.2.1. Como as regras do `SELECT` funcionam

As regras `ON SELECT` são aplicadas a todos os comandos como o último passo, mesmo que o comando seja um `INSERT`, `UPDATE` ou `DELETE`, e possuem semântica diferente das regras para os outros tipos de comando, porque modificam a árvore de comando diretamente em vez de criar uma nova. Portanto as regras do `SELECT` são descritas primeiro.

Atualmente só pode haver uma ação em uma regra `ON SELECT`, e deve ser uma ação `SELECT` incondicional que seja `INSTEAD`. Esta restrição é necessária para tornar as regras seguras o suficiente para serem abertas aos usuários comuns, restringindo as regras `ON SELECT` a agirem como visões.

Os exemplos deste capítulo são duas visões com junção que realizam alguns cálculos, e mais algumas visões que utilizam estas visões. Uma das duas primeiras visões é personalizada posteriormente adicionando regras para as operações `INSERT`, `UPDATE` e `DELETE`, para que o resultado final seja uma visão que se comporte como uma tabela real com alguma funcionalidade mágica. Não é um exemplo simples para se começar, torna o assunto mais difícil de ser entendido, mas é melhor ter um exemplo que cubra todos os pontos discutido passo a passo, do que ter muitos exemplos diferentes o que pode acabar confundindo a mente.

Para o exemplo é necessária uma pequena função, `min`, que retorna o menor entre dois valores inteiros. É criada como:

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS $$
    SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END
$$ LANGUAGE SQL STRICT;
```

As tabelas reais necessárias nas duas primeiras descrições do sistemas de regras são estas:

```
CREATE TABLE tbl_sapato (
    sap_nome          text,          -- nome do sapato - chave primária
    sap_num_par_disp  integer,       -- número de pares disponíveis
    sap_cor_cad_pref  text,          -- cor preferida do cadarço do sapato
    sap_comp_cad_min   real,          -- comprimento mínimo do cadarço do sapato
    sap_comp_cad_max   real,          -- comprimento máximo do cadarço do sapato
    sap_comp_cad_unid  text          -- unidade de comprimento
);

CREATE TABLE tbl_cadarço (
    cad_sap_nome      text,          -- nome do cadarço do sapato - chave primária
    cad_sap_num_par_disp integer,    -- número de pares disponíveis
    cad_sap_cor        text,          -- cor do cadarço
    cad_sap_comp        real,          -- comprimento do cadarço
    cad_sap_unid        text          -- unidade de comprimento
);

CREATE TABLE tbl_unidade (
    uni_nome          text,          -- nome da unidade - chave primária
    uni_fator          real          -- fator para transformar em cm
);
```

Como pode ser visto, representa os dados de uma loja de sapatos.

As visões são criadas como

```
CREATE VIEW vis_sapato AS
    SELECT sap.sap_nome,
           sap.sap_num_par_disp,
           sap.sap_cor_cad_pref,
           sap.sap_comp_cad_min,
           sap.sap_comp_cad_min * u.uni_fator AS sap_comp_cad_min_cm,
           sap.sap_comp_cad_max,
```

```

        sap.sap_comp_cad_max * u.uni_fator AS sap_comp_cad_max_cm,
        sap.sap_comp_cad_unid
    FROM tbl_sapato sap, tbl_unidade u
    WHERE sap.sap_comp_cad_unid = u.uni_nome;

CREATE VIEW vis_cadarço AS
    SELECT cad.cad_sap_nome,
           cad.cad_sap_num_par_disp,
           cad.cad_sap_cor,
           cad.cad_sap_comp,
           cad.cad_sap_unid,
           cad.cad_sap_comp * uni.uni_fator AS cad_sap_comp_cm
    FROM tbl_cadarço cad, tbl_unidade uni
    WHERE cad.cad_sap_unid = uni.uni_nome;

CREATE VIEW vis_sapato_pronto AS
    SELECT vsap.sap_nome,
           vsap.sap_num_par_disp,
           vcad.cad_sap_nome,
           vcad.cad_sap_num_par_disp,
           min(vsap.sap_num_par_disp, vcad.cad_sap_num_par_disp) AS total_disp
    FROM vis_sapato vsap, vis_cadarço vcad
    WHERE vcad.cad_sap_cor = vsap.sap_cor_cad_pref
           AND vcad.cad_sap_comp_cm >= vsap.sap_comp_cad_min_cm
           AND vcad.cad_sap_comp_cm <= vsap.sap_comp_cad_max_cm;

```

O comando `CREATE VIEW` para a visão `vis_cadarço` (que é a mais simples que temos) cria a relação `vis_cadarço`, e uma entrada em `pg_rewrite` informando que existe uma regra de reescrita que deve ser aplicada sempre que a relação `vis_cadarço` for referenciada na tabela de abrangência do comando. A regra não possui nenhuma qualificação de regra (discutido mais tarde, nas regras não-`SELECT`, uma vez que atualmente as regras `SELECT` não podem tê-las), e é do tipo `INSTEAD`. Deve ser observado que qualificação de regra não é o mesmo que qualificação de comando. A ação da nossa regra possui uma qualificação de comando. A ação da regra é uma árvore de comando que é a cópia da instrução `SELECT` do comando de criação da visão.

**Nota:** As duas entradas adicionais na tabela de abrangência para `NEW` e `OLD` (chamadas na árvore de comando impressa de `*NEW*` e `*OLD*` por motivos históricos) que podem ser vistas em `pg_rewrite` não são de interesse para as regras de `SELECT`.

Agora são inseridas linhas nas tabelas `tbl_unidade`, `tbl_sapato` e `tbl_cadarço`, e executada uma consulta simples na visão:

```

INSERT INTO tbl_unidade VALUES ('cm', 1.0);
INSERT INTO tbl_unidade VALUES ('m', 100.0);
INSERT INTO tbl_unidade VALUES ('inch', 2.54);

INSERT INTO tbl_sapato VALUES ('sap1', 2, 'preto', 70.0, 90.0, 'cm');
INSERT INTO tbl_sapato VALUES ('sap2', 0, 'preto', 30.0, 40.0, 'inch');
INSERT INTO tbl_sapato VALUES ('sap3', 4, 'marrom', 50.0, 65.0, 'cm');
INSERT INTO tbl_sapato VALUES ('sap4', 3, 'marrom', 40.0, 50.0, 'inch');

INSERT INTO tbl_cadarço VALUES ('cad1', 5, 'preto', 80.0, 'cm');
INSERT INTO tbl_cadarço VALUES ('cad2', 6, 'preto', 100.0, 'cm');
INSERT INTO tbl_cadarço VALUES ('cad3', 0, 'preto', 35.0, 'inch');
INSERT INTO tbl_cadarço VALUES ('cad4', 8, 'preto', 40.0, 'inch');
INSERT INTO tbl_cadarço VALUES ('cad5', 4, 'marrom', 1.0, 'm');
INSERT INTO tbl_cadarço VALUES ('cad6', 0, 'marrom', 0.9, 'm');
INSERT INTO tbl_cadarço VALUES ('cad7', 7, 'marrom', 60, 'cm');
INSERT INTO tbl_cadarço VALUES ('cad8', 1, 'marrom', 40, 'inch');

SELECT * FROM vis_cadarço;

```

cad_sap_nome	cad_sap_num_par_disp	cad_sap_cor	cad_sap_comp	cad_sap_unid	cad_sap_comp_cm
cad1	5	preto	80	cm	80
cad2	6	preto	100	cm	100
cad7	7	marrom	60	cm	60
cad3	0	preto	35	inch	88.9
cad4	8	preto	40	inch	101.6
cad8	1	marrom	40	inch	101.6
cad5	4	marrom	1	m	100
cad6	0	marrom	0.9	m	90

(8 linhas)

Este é o `SELECT` mais simples que pode ser feito nestas visões, portanto é utilizada esta oportunidade para explicar os princípios básicos das regras de visão. A instrução `SELECT * FROM vis_cadarço` após ser interpretada pelo analisador produz a árvore de comando

```
SELECT vis_cadarço.cad_sap_nome, vis_cadarço.cad_sap_num_par_disp,
       vis_cadarço.cad_sap_cor, vis_cadarço.cad_sap_comp,
       vis_cadarço.cad_sap_unid, vis_cadarço.cad_sap_comp_cm
FROM vis_cadarço vis_cadarço;
```

entregue ao sistema de regras. O sistema de regras percorre a tabela de abrangência e verifica se há regras para alguma relação. Ao processar a entrada para `vis_cadarço` (a única até agora) na tabela de abrangência encontra a regra `_RETURN` com a árvore de comando

```
SELECT cad.cad_sap_nome, cad.cad_sap_num_par_disp,
       cad.cad_sap_cor, cad.cad_sap_comp, cad.cad_sap_unid,
       cad.cad_sap_comp * uni.uni_fator AS cad_sap_comp_cm
FROM vis_cadarço *OLD*, vis_cadarço *NEW*,
     tbl_cadarço cad, tbl_unidade uni
WHERE cad.cad_sap_unid = uni.uni_nome;
```

Para expandir a visão o reescritor simplesmente cria uma entrada na tabela de abrangência do subcomando, contendo a árvore de comando da ação da regra, e substitui esta entrada da tabela de abrangência pela original que fazia referência à visão. A árvore de comando reescrita resultante é praticamente a mesma que seria se tivesse sido digitado

```
SELECT vis_cadarço.cad_sap_nome, vis_cadarço.cad_sap_num_par_disp,
       vis_cadarço.cad_sap_cor, vis_cadarço.cad_sap_comp,
       vis_cadarço.cad_sap_unid, vis_cadarço.cad_sap_comp_cm
FROM (SELECT cad.cad_sap_nome,
            cad.cad_sap_num_par_disp,
            cad.cad_sap_cor,
            cad.cad_sap_comp,
            cad.cad_sap_unid,
            cad.cad_sap_comp * uni.uni_fator AS cad_sap_comp_cm
      FROM tbl_cadarço cad, tbl_unidade uni
      WHERE cad.cad_sap_unid = uni.uni_nome) vis_cadarço;
```

Entretanto, há uma diferença: a tabela de abrangência do subcomando possui duas entradas adicionais, `vis_cadarço *OLD*` e `vis_cadarço *NEW*`. Estas entradas não participam diretamente no comando, uma vez que não são referenciadas pela árvore de junção ou pela lista de destino do subcomando. O reescritor utiliza as mesmas para armazenar informações de verificação de privilégio de acesso presentes originalmente na entrada da tabela de abrangência que fazia referência à visão. Desta maneira, o executor ainda vai verificar se o usuário possui os privilégios apropriados para acessar a visão, muito embora não exista uso direto da visão no comando reescrito.

Esta foi a aplicação da primeira regra. O sistema de regras continua verificando as entradas remanescentes na tabela de abrangência do comando no topo (neste exemplo não há mais nenhuma), e verifica recursivamente as entradas nas tabelas de abrangência dos subcomandos adicionados para verificar se alguma destes faz referência a visão (Mas não expande `*OLD*` ou `*NEW*` — senão haveria uma recursão infinita!) Neste exemplo não existem regras de reescrita para `tbl_cadarço` ou `tbl_unidade`. Portanto a reescrita está completa e a instrução acima é o resultado final entregue ao planejador.

Agora desejamos escrever uma consulta para descobrir para quais sapatos na loja existem cadarços correspondentes (cor e comprimento), e com número total de pares com correspondência exata maior ou igual a dois.

```
SELECT * FROM vis_sapato_pronto WHERE total_disp >= 2;
```

sap_nome	sap_num_par_disp	cad_sap_nome	cad_sap_num_par_disp	total_disp
sap1	2	cad1	5	2
sap3	4	cad7	7	4

(2 linhas)

Desta vez a saída do analisador é a árvore de comando

```
SELECT vis_sapato_pronto.sap_nome, vis_sapato_pronto.sap_num_par_disp,
       vis_sapato_pronto.cad_sap_nome, vis_sapato_pronto.cad_sap_num_par_disp,
       vis_sapato_pronto.total_disp
FROM vis_sapato_pronto vis_sapato_pronto
WHERE vis_sapato_pronto.total_disp >= 2;
```

A primeira regra aplicada será a da visão vis\_sapato\_pronto, resultando na árvore de comando

```
SELECT vis_sapato_pronto.sap_nome, vis_sapato_pronto.sap_num_par_disp,
       vis_sapato_pronto.cad_sap_nome, vis_sapato_pronto.cad_sap_num_par_disp,
       vis_sapato_pronto.total_disp
FROM (SELECT vsap.sap_nome,
            vsap.sap_num_par_disp,
            vcad.cad_sap_nome,
            vcad.cad_sap_num_par_disp,
            min(vsap.sap_num_par_disp, vcad.cad_sap_num_par_disp) AS total_disp
      FROM vis_sapato vsap, vis_cadarço vcad
      WHERE vcad.cad_sap_cor = vsap.sap_cor_cad_pref
            AND vcad.cad_sap_comp_cm >= vsap.sap_comp_cad_min_cm
            AND vcad.cad_sap_comp_cm <= vsap.sap_comp_cad_max_cm) vis_sapato_pronto
WHERE vis_sapato_pronto.total_disp >= 2;
```

De maneira semelhante, as regras para vis\_sapato e vis\_cadarço são substituídas na tabela de abrangência do subcomando, conduzindo a uma árvore de comando final com três níveis:

```
SELECT vis_sapato_pronto.sap_nome, vis_sapato_pronto.sap_num_par_disp,
       vis_sapato_pronto.cad_sap_nome, vis_sapato_pronto.cad_sap_num_par_disp,
       vis_sapato_pronto.total_disp
FROM (SELECT vsap.sap_nome,
            vsap.sap_num_par_disp,
            vcad.cad_sap_nome,
            vcad.cad_sap_num_par_disp,
            min(vsap.sap_num_par_disp, vcad.cad_sap_num_par_disp) AS total_disp
      FROM (SELECT sap.sap_nome,
                  sap.sap_num_par_disp,
                  sap.sap_cor_cad_pref,
                  sap.sap_comp_cad_min,
                  sap.sap_comp_cad_min * u.uni_fator AS sap_comp_cad_min_cm,
                  sap.sap_comp_cad_max,
                  sap.sap_comp_cad_max * u.uni_fator AS sap_comp_cad_max_cm,
                  sap.sap_comp_cad_unid
            FROM tbl_sapato sap, tbl_unidade u
            WHERE sap.sap_comp_cad_unid = u.uni_nome) vsap,
      (SELECT cad.cad_sap_nome,
            cad.cad_sap_num_par_disp,
            cad.cad_sap_cor,
            cad.cad_sap_comp,
            cad.cad_sap_unid,
            cad.cad_sap_comp * uni.uni_fator AS cad_sap_comp_cm
```

```

        FROM tbl_cadarço cad, tbl_unidade uni
        WHERE cad.cad_sap_unid = uni.uni_nome) vcad
    WHERE vcad.cad_sap_cor = vsap.sap_cor_cad_pref
        AND vcad.cad_sap_comp_cm >= vsap.sap_comp_cad_min_cm
        AND vcad.cad_sap_comp_cm <= vsap.sap_comp_cad_max_cm) vis_sapato_pronto
    WHERE vis_sapato_pronto.total_disp >= 2;

```

Acontece que o planejador colapsa esta árvore em uma árvore de comando de dois níveis: os comandos `SELECT` na parte inferior são “puxados” para o `SELECT` do meio, uma vez que não é necessário processá-los separadamente. Porém, o `SELECT` do meio permanece separado do de cima, porque contém funções de agregação. Se fosse puxado para cima mudaria do comportamento do `SELECT` do topo, o que não se deseja. Entretanto, colapsar a árvore de comando é uma otimização que o sistema de reescrita não tem que se preocupar a mesma.

**Nota:** Atualmente não existe no sistema de regras mecanismo para interromper a recursão das regras de visão (somente para os outros tipos de regra). Isto não é um problema sério, porque a única maneira de provocar um laço sem fim (inchando o processo servidor até que este chegue ao limite de memória), é criar as tabelas e depois definir as regras de visão manualmente utilizando `CREATE RULE`, de uma maneira que a primeira tabela selecione da segunda e a segunda selecione da primeira. Esta situação não pode acontecer quando se utiliza `CREATE VIEW`, porque no primeiro `CREATE VIEW` a segunda relação não existe e, portanto, a primeira relação não pode selecionar da segunda.

### 33.2.2. Regras de visão em instruções não-SELECT

Dois detalhes da árvore de comando não foram tocados na descrição das regras de visão acima. São estes o tipo do comando e a relação do resultado. De fato, as regras de visão não precisam desta informação.

Existem poucas diferenças entre uma árvore de comando para o `SELECT` e para qualquer outro comando. Obviamente possuem tipos de comando diferentes e, fora o `SELECT`, a relação do resultado aponta para uma entrada na tabela de abrangência para onde o resultado deve ir. Tudo mais é exatamente o mesmo. Portanto, se existirem as tabelas `t1` e `t2` com as colunas `a` e `b`, as árvores de comando para as instruções

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

```
UPDATE t1 SET b = t2.b WHERE t1.a = t2.a;
```

são praticamente idênticas. Em particular:

- As tabelas de abrangência possuem entradas para as tabelas `t1` e `t2`.
- As listas de destino contêm uma variável que aponta para a coluna `b` da entrada na tabela de abrangência para a tabela `t2`.
- As expressões de qualificação comparam as colunas `a` das duas entradas na tabela de abrangência com relação a igualdade.
- As árvores de junção mostram uma junção simples entre `t1` e `t2`.

A consequência é que as duas árvores de comando resultam em planos de execução semelhantes: ambos são junções de duas tabelas. Para o `UPDATE` as colunas de `t1` que faltam são adicionadas à lista de destino pelo planejador, e a árvore de comando final fica sendo:

```
UPDATE t1 SET a = t1.a, b = t2.b WHERE t1.a = t2.a;
```

Portanto, o processamento do executor sobre a junção produz exatamente o mesmo conjunto de resultados que

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

produz, mas existe um pequeno problema no `UPDATE`: para o executor não interessa para que serve o resultado da junção sendo feita. Apenas produz um conjunto de linhas de resultado. A diferença que um comando é `SELECT` e que o outro é `UPDATE` é tratado por quem chama o executor. Quem chama ainda sabe (olhando na árvore de comando) que este comando é um `UPDATE`, e sabe que este resultado deve ir para a tabela `t1`. Mas qual das linhas presentes deve ser substituída pela nova linha?

Para resolver este problema é adicionada uma outra entrada na lista de destino da instrução `UPDATE` (e também da instrução `DELETE`): o identificador da tupla corrente (CTID). Esta é uma coluna do sistema contendo o número de bloco do

arquivo, e a posição da linha no bloco. Sabendo a tabela, o CTID pode ser utilizado para trazer a linha original de `t1` a ser atualizada. Após adicionar CTID à lista de destino, o comando se parece com:

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Agora entra em cena um outro detalhe do PostgreSQL. As linhas antigas da tabela não são sobrescritas, e por causa disto o `ROLLBACK` é rápido. Em uma instrução `UPDATE` a nova linha de resultado é inserida na tabela (após eliminar o CTID), e no cabeçalho de linha da linha antiga, para o qual CTID apontava, as entradas `cmax` e `xmax` são definidas como o contador de comando corrente e identificador de transação corrente. Portanto a linha antiga fica escondida, e após a efetivação da transação o comando `VACUUM` pode remover a linha.

Sabendo disso tudo, pode-se simplesmente aplicar as regras de visão exatamente da mesma maneira para qualquer comando. Não existe diferença.

### 33.2.3. O poder das visões no PostgreSQL

O que foi visto acima demonstra como o sistema de regras incorpora as definições de visão na árvore de comando original. No segundo exemplo um simples `SELECT` de uma visão criou uma árvore de comando final que é a junção de quatro tabelas (`tbl_unidade` foi utilizada duas vezes com nomes diferentes).

O benefício de implementar as visões pelo sistema de regras é que o planejador possui todas as informações sobre quais tabelas devem ser varridas, mais o relacionamento entre estas tabelas, mais as qualificações restritivas das visões, mais as qualificações do comando original, em uma única árvore de comando. Esta permanece sendo a situação quando o comando original já é uma junção de visões. O planejador tem que decidir qual o melhor caminho para executar o comando, e quanto mais informações o planejador tiver melhor poderá ser a decisão. A forma de implementação do sistema de regras no PostgreSQL garante que esta é toda a informação disponível sobre o comando até este ponto.

### 33.2.4. Atualização de visão

O que acontece se uma visão for nomeada como a relação de destino de uma instrução `INSERT`, `UPDATE` ou `DELETE`? Após serem feitas as substituições descritas acima, será obtida uma árvore de comando na qual a relação do resultado aponta para uma entrada na tabela de abrangência do subcomando. Isto não funciona e, portanto, o reescritor lança um erro quando vê que produziu algo deste tipo.

Para se mudar esta situação, podem ser definidas regras que modificam o comportamento destes tipos de comando. Este é o assunto da próxima seção.

## 33.3. Regras para INSERT, UPDATE e DELETE

As regras definidas para `INSERT`, `UPDATE` e `DELETE` são significativamente diferentes das regras de visão descritas na seção anterior. Em primeiro lugar, porque os comandos `CREATE RULE` destas regras permitem mais opções:

- É permitido que não tenham nenhuma ação.
- É permitido que tenham várias ações.
- Podem ser `INSTEAD` (em vez de) ou `ALSO` (também) (padrão).
- As pseudorelações `NEW` e `OLD` se tornam úteis.
- Podem ter qualificação de regras.

Em segundo lugar, não modificam a árvore de comando diretamente. Em vez disso, criam zero ou mais árvores de comando novas, e podem abandonar a árvore original.

### 33.3.1. Como as regras de atualização funcionam

Tenha em mente a sintaxe

```
CREATE RULE nome_da_regra AS ON evento
    TO objeto [WHERE qualificação_da_regra]
    DO [ALSO|INSTEAD] [ação | (ações) | NOTHING];
```

No que vem a seguir, *regras de atualização* significa regras definidas para `INSERT`, `UPDATE` ou `DELETE`.

As regras de atualização são aplicadas pelo sistema de regras quando a relação do resultado e o tipo de comando da árvore de comando são iguais ao objeto e evento especificados no comando `CREATE RULE`. Para as regras de atualização o sistema cria uma lista de árvores do comando. Inicialmente a lista de árvores de comando está vazia. Podem haver zero (palavra chave `NOTHING`), uma, ou várias ações. Para simplificar, será vista uma regra com uma ação. Esta regra pode ter uma qualificação, ou não, e pode ser `INSTEAD` ou `ALSO` (padrão).

O que é uma qualificação de regra? É uma restrição que informa quando as ações da regra devem ser realizadas e quando não devem. Esta qualificação somente pode fazer referência às pseudorelações `NEW` e/ou `OLD`, que representam basicamente a relação fornecida como objeto (mas com um significado especial).

Portanto tem-se quatro casos que produzem as seguintes árvores de comando para uma regra de uma ação.

Sem qualificação e `ALSO`

a árvore de comando da ação da regra com a qualificação da árvore de comando original adicionada

Sem qualificação mas `INSTEAD`

a árvore de comando da ação da regra com a qualificação da árvore de comando original adicionada

Qualificação fornecida e `ALSO`

a árvore de comando da ação da regra com a qualificação da regra e a qualificação da árvore de comando original adicionada

Qualificação fornecida e `INSTEAD`

a árvore de comando da ação da regra com a qualificação da regra e a qualificação da árvore de comando original; e a árvore de comando original com a qualificação da regra negada adicionada.

Por fim, se a regra for `ALSO`, a árvore de comando original não modificada é adicionada à lista. Uma vez que somente as regras `INSTEAD` qualificadas adicionam a árvore de comando original, acaba-se com uma ou duas árvores de comando de saída em uma regra com uma ação.

Para as regras `ON INSERT`, o comando original (se não for suprimido pelo `INSTEAD`) é executado antes de qualquer ação adicionada pelas regras. Isto permite as ações enxergarem as linhas inseridas. Porém, para as regras `ON UPDATE` e `ON DELETE` o comando original é executado após as ações adicionadas pelas regras. Isto garante que as ações podem enxergar as linhas a serem atualizadas ou excluídas; caso contrário, as ações não podem fazer nada porque não encontram linhas correspondendo às suas qualificações.

As árvores de comando geradas a partir das ações das regras são lançadas no sistema de reescrita novamente, e talvez sejam aplicadas mais regras resultando em um número maior, ou menor, de árvores de comando. Portanto, as árvores de comando nas ações das regras devem ter um tipo de comando diferente ou uma relação do resultado diferente, senão este processo recursivo se torna um laço. Atualmente existe um limite de recursão estabelecido em 100 interações. Se após 100 interações ainda existirem regras de atualização a serem aplicadas, o sistema de regras assume que está ocorrendo um laço sobre várias definições de regra e relata um erro.

As árvores de comando encontradas nas ações do catálogo do sistema `pg_rewrite` são somente modelos (`templates`). Uma vez que podem fazer referência às entradas `NEW` e `OLD` da tabela de abrangência, devem ser feitas algumas substituições antes que possa ser utilizadas. Para toda referência a `NEW`, é procurado na lista de destino do comando original uma entrada correspondente. Se for encontrada, a expressão desta entrada substitui a referência. Senão, `NEW` significa o mesmo que `OLD` (para o `UPDATE`), ou é substituído por um valor nulo (para o `INSERT`). Toda referência a `OLD` é substituída por uma referência a uma entrada na tabela de abrangência que é a relação do resultado.

Após o sistema terminar de aplicar as regras de atualização, aplica as regras de visão às árvores de comando produzidas. As visões não podem inserir novas ações de atualização, portanto não é necessário aplicar regras de atualização à saída da reescrita da regra.

### 33.3.1.1. Uma primeira regra passo-a-passo

Digamos que se deseja acompanhar as alterações na coluna `cad_sap_num_par_disp` da relação `tbl_cadarço`. Para essa finalidade é criada uma tabela de acompanhamento, e uma regra que escreve sob condição uma entrada de acompanhamento quando é executada uma atualização na tabela `tbl_cadarço`.

```
CREATE TABLE tbl_cadarço_log (
    cad_sap_nome      text,      -- nome do cadarço do sapato
```



```

    cad_sap_num_par_disp integer, -- novo valor disponível
    log_quem             text,    -- quem fez isto
    log_quando           timestamp -- quando
);

CREATE RULE reg_cadarço_upd AS ON UPDATE TO tbl_cadarço
WHERE NEW.cad_sap_num_par_disp <> OLD.cad_sap_num_par_disp
DO INSERT INTO tbl_cadarço_log VALUES (
    NEW.cad_sap_nome,
    NEW.cad_sap_num_par_disp,
    current_user,
    current_timestamp
);

```

Depois disso, se for executado

```
UPDATE tbl_cadarço SET cad_sap_num_par_disp = 6 WHERE cad_sap_nome = 'cad7';
```

e olhada a tabela de acompanhamento, será encontrado:

```
SELECT * FROM tbl_cadarço_log;
```

cad_sap_nome	cad_sap_num_par_disp	log_quem	log_quando
cad7	6	teste	2005-12-03 07:45:28.500131

(1 linha)

Que é o esperado. O que aconteceu em segundo plano foi o seguinte: O analisador criou a árvore de comando

```

UPDATE tbl_cadarço SET cad_sap_num_par_disp = 6
FROM tbl_cadarço tbl_cadarço
WHERE tbl_cadarço.cad_sap_nome = 'cad7';

```

e existe a regra `reg_cadarço_upd`, que é do tipo `ON UPDATE`, contendo a expressão de qualificação de regra

```
NEW.cad_sap_num_par_disp <> OLD.cad_sap_num_par_disp
```

e a ação

```

INSERT INTO tbl_cadarço_log VALUES (
    *NEW*.cad_sap_nome, *NEW*.cad_sap_num_par_disp,
    current_user, current_timestamp )
FROM tbl_cadarço *NEW*, tbl_cadarço *OLD*;

```

(Isto parece um pouco estranho, uma vez que normalmente não se pode escrever `INSERT ... VALUES ... FROM`. Neste caso, a cláusula `FROM` serve apenas para indicar que existem entradas na tabela de abrangência da árvore de comando para `*NEW* *OLD*`. São necessárias para que possam ser referenciadas pelas variáveis na árvore de comando do `INSERT`)

A regra é uma regra `ALSO` qualificada, portanto o sistema de regras precisa retornar duas árvores de comando: a ação da regra modificada e a árvore de comando original. No passo 1, a tabela de abrangência do comando original é incorporada à árvore de comando da ação da regra, resultando em:

```

INSERT INTO tbl_cadarço_log VALUES (
    *NEW*.cad_sap_nome, *NEW*.cad_sap_num_par_disp,
    current_user, current_timestamp )
FROM tbl_cadarço *NEW*, tbl_cadarço *OLD*,
    tbl_cadarço tbl_cadarço;

```

No passo 2, a qualificação da regra é adicionada, ficando o conjunto de resultados restrito às linhas onde `cad_sap_num_par_disp` muda de valor:

```

INSERT INTO tbl_cadarço_log VALUES (
    *NEW*.cad_sap_nome, *NEW*.cad_sap_num_par_disp,

```

```

current_user, current_timestamp )
FROM tbl_cadarço *NEW*, tbl_cadarço *OLD*,
tbl_cadarço tbl_cadarço
WHERE *NEW*.cad_sap_num_par_disp <> *OLD*.cad_sap_num_par_disp;

```

(Isto parece ainda mais estranho, uma vez que INSERT ... VALUES também não tem uma cláusula a WHERE, mas o planejador e o executor não têm dificuldade para lidar com esta situação. De qualquer forma precisam dar suporte a esta mesma funcionalidade para INSERT ... SELECT.)

No passo 3 é adicionada a qualificação da árvore de comando original, restringindo o conjunto de resultados ainda mais, para somente as linhas afetadas pelo comando original:

```

INSERT INTO tbl_cadarço_log VALUES (
    *NEW*.cad_sap_nome, *NEW*.cad_sap_num_par_disp,
    current_user, current_timestamp )
FROM tbl_cadarço *NEW*, tbl_cadarço *OLD*,
tbl_cadarço tbl_cadarço
WHERE *NEW*.cad_sap_num_par_disp <> *OLD*.cad_sap_num_par_disp
AND tbl_cadarço.cad_sap_nome = 'cad7';

```

O passo 4 substitui as referências a NEW pelas entradas na lista de destino da árvore de comando original, ou pelas referências à variável correspondente da relação do resultado:

```

INSERT INTO tbl_cadarço_log VALUES (
    tbl_cadarço.cad_sap_nome, 6,
    current_user, current_timestamp )
FROM tbl_cadarço *NEW*, tbl_cadarço *OLD*,
tbl_cadarço tbl_cadarço
WHERE 6 <> *OLD*.cad_sap_num_par_disp
AND tbl_cadarço.cad_sap_nome = 'cad7';

```

O passo 5 troca as referências a OLD por referências a relação do resultado:

```

INSERT INTO tbl_cadarço_log VALUES (
    tbl_cadarço.cad_sap_nome, 6,
    current_user, current_timestamp )
FROM tbl_cadarço *NEW*, tbl_cadarço *OLD*,
tbl_cadarço tbl_cadarço
WHERE 6 <> tbl_cadarço.cad_sap_num_par_disp
AND tbl_cadarço.cad_sap_nome = 'cad7';

```

Está pronto. Uma vez que a regra é ALSO, a árvore de comando original também é enviada para a saída. Em resumo, a saída do sistema de regras é uma lista com duas árvores de comando que correspondem a estas instruções:

```

INSERT INTO tbl_cadarço_log VALUES (
    tbl_cadarço.cad_sap_nome, 6,
    current_user, current_timestamp )
FROM tbl_cadarço
WHERE 6 <> tbl_cadarço.cad_sap_num_par_disp
AND tbl_cadarço.cad_sap_nome = 'cad7';

```

```

UPDATE tbl_cadarço SET cad_sap_num_par_disp = 6
WHERE cad_sap_nome = 'cad7';

```

São executadas nesta ordem, e é exatamente isto o que a regra deveria fazer.

As substituições e as qualificações adicionadas garantem que se o comando original fosse, digamos,

```

UPDATE tbl_cadarço SET cad_sap_cor = 'verde'
WHERE cad_sap_nome = 'cad7';

```

não seria escrito nenhuma entrada de acompanhamento. Neste caso, a árvore de comando original não contém a entrada na lista de destino para cad\_sap\_num\_par\_disp, portanto NEW.cad\_sap\_num\_par\_disp é substituído por tbl\_cadarço.cad\_sap\_num\_par\_disp. Portanto, o comando extra gerado por esta regra é

```
INSERT INTO tbl_cadarço_log VALUES (
    tbl_cadarço.cad_sap_nome, tbl_cadarço.cad_sap_num_par_disp,
    current_user, current_timestamp )
FROM tbl_cadarço
WHERE tbl_cadarço.cad_sap_num_par_disp <> tbl_cadarço.cad_sap_num_par_disp
AND tbl_cadarço.cad_sap_nome = 'cad7';
```

e a qualificação nunca será verdade.

A regra também funciona quando o comando original modifica várias linhas. Portanto, se for executado o comando

```
UPDATE tbl_cadarço SET cad_sap_num_par_disp = 0
WHERE cad_sap_cor = 'preto';
```

de fato serão atualizadas quatro linhas (cad1, cad2, cad3 e cad4), mas cad3 já tem cad\_sap\_num\_par\_disp = 0. Neste caso, a qualificação original das árvores de comando é diferente, e isto resulta na geração da árvore de comando adicional

```
INSERT INTO tbl_cadarço_log
SELECT tbl_cadarço.cad_sap_nome, 0,
    current_user, current_timestamp
FROM tbl_cadarço
WHERE 0 <> tbl_cadarço.cad_sap_num_par_disp
AND tbl_cadarço.cad_sap_cor = 'preto';
```

pela regra. Esta árvore de comando com certeza insere três novas entradas de acompanhamento. E isto está inteiramente correto.

Aqui pode ser visto porque é importante a árvore de comando original ser executada por último. Se o UPDATE tivesse sido executado primeiro, todos os valores das linhas já teriam sido definidos como zero e, portanto, o acompanhamento do INSERT não encontraria uma linha onde 0 <> tbl\_cadarço.cad\_sap\_num\_par\_disp.

### 33.3.2. Cooperação com visões

Uma forma simples de proteger as relações das visões contra a possibilidade mencionada de alguém tentar executar os comandos INSERT, UPDATE ou DELETE nas mesmas, é deixando estas árvores de comando serem jogadas fora. Para isso são criadas as regras

```
CREATE RULE vis_sapato_ins_protege AS ON INSERT TO vis_sapato
DO INSTEAD NOTHING;
CREATE RULE vis_sapato_upd_protege AS ON UPDATE TO vis_sapato
DO INSTEAD NOTHING;
CREATE RULE vis_sapato_del_protege AS ON DELETE TO vis_sapato
DO INSTEAD NOTHING;
```

Depois disso, se alguém tentar fazer uma destas operações na relação da visão vis\_sapato, o sistema de regras aplica estas regras. Uma vez que as regras não possuem ação e são INSTEAD, a lista de árvores de comando resultante estará vazia, e todo o comando se transforma em nada, porque não há nada deixado para ser otimizado ou executado após o sistema de regras terminar de executar.

Uma utilização mais sofisticadas do sistema de regras, é para criar regras que reescrevem uma árvore de comando em outra que realiza a operação correta nas tabelas de verdade. Para se fazer isto na visão vis\_cadarço, são criadas as seguintes regras:

```
CREATE RULE vis_cadarço_ins AS ON INSERT TO vis_cadarço
DO INSTEAD
INSERT INTO tbl_cadarço VALUES (
    NEW.cad_sap_nome,
    NEW.cad_sap_num_par_disp,
    NEW.cad_sap_cor,
    NEW.cad_sap_comp,
    NEW.cad_sap_unid
);
```

```

CREATE RULE vis_cadarço_upd AS ON UPDATE TO vis_cadarço
DO INSTEAD
UPDATE tbl_cadarço
SET cad_sap_nome = NEW.cad_sap_nome,
    cad_sap_num_par_disp = NEW.cad_sap_num_par_disp,
    cad_sap_cor = NEW.cad_sap_cor,
    cad_sap_comp = NEW.cad_sap_comp,
    cad_sap_unid = NEW.cad_sap_unid
WHERE cad_sap_nome = OLD.cad_sap_nome;

CREATE RULE vis_cadarço_del AS ON DELETE TO vis_cadarço
DO INSTEAD
DELETE FROM tbl_cadarço
WHERE cad_sap_nome = OLD.cad_sap_nome;

```

Agora é assumido que, de vez em quando, chega na loja um pacote de cadarços e uma longa lista de partes junto com este, mas que não se deseja atualizar manualmente a visão `vis_cadarço` toda vez que chega um destes pacotes. Para isso são criadas duas pequenas tabelas: uma é onde são inseridos os itens da lista de partes, e outra com um truque especial. Os comandos de criação destas tabelas são:

```

CREATE TABLE tbl_cadarço_chegada (
    cheg_nome    text,
    cheg_quant   integer
);

CREATE TABLE tbl_cadarço_ok (
    ok_nome      text,
    ok_quant     integer
);

CREATE RULE tbl_cadarço_ok_ins AS ON INSERT TO tbl_cadarço_ok
DO INSTEAD
UPDATE vis_cadarço
SET cad_sap_num_par_disp = cad_sap_num_par_disp + NEW.ok_quant
WHERE cad_sap_nome = NEW.ok_nome;

```

Agora a tabela `tbl_cadarço_chegada` pode ser preenchida com os dados da lista de partes:

```

INSERT INTO tbl_cadarço_chegada VALUES('cad3',10);
INSERT INTO tbl_cadarço_chegada VALUES('cad6',20);
INSERT INTO tbl_cadarço_chegada VALUES('cad8',20);

```

```
SELECT * FROM tbl_cadarço_chegada;
```

cheg_nome	cheg_quant
sap3	10
sap6	20
sap8	20

(3 linhas)

Dando uma olhada rápida nos dados atuais encontramos:

```
SELECT * FROM vis_cadarço;
```

cad_sap_nome	cad_sap_num_par_disp	cad_sap_cor	cad_sap_comp	cad_sap_unid	cad_sap_comp_cm
cad1	5	preto	80	cm	80
cad2	6	preto	100	cm	100
cad7	6	marrom	60	cm	60
cad3	0	preto	35	inch	88.9

cad4		8		preto		40		inch		101.6
cad8		1		marrom		40		inch		101.6
cad5		4		marrom		1		m		100
cad6		0		marrom		0.9		m		90

(8 linhas)

Agora os cadarços que chegaram são movidos com:

```
INSERT INTO tbl_cadarço_ok SELECT * FROM tbl_cadarço_chegada;
```

e verificado os resultados:

```
SELECT * FROM vis_cadarço ORDER BY cad_sap_nome;
```

cad_sap_nome	cad_sap_num_par_disp	cad_sap_cor	cad_sap_comp	cad_sap_unid	cad_sap_comp_cm
cad1	5	preto	80	cm	80
cad2	6	preto	100	cm	100
cad3	10	preto	35	inch	88.9
cad4	8	preto	40	inch	101.6
cad5	4	marrom	1	m	100
cad6	20	marrom	0.9	m	90
cad7	6	marrom	60	cm	60
cad8	21	marrom	40	inch	101.6

(8 linhas)

```
SELECT * FROM tbl_cadarço_log;
```

cad_sap_nome	cad_sap_num_par_disp	log_quem	log_quando
cad7	6	teste	2005-12-03 08:31:22.822485
cad3	10	teste	2005-12-03 08:37:15.497084
cad6	20	teste	2005-12-03 08:37:15.497084
cad8	21	teste	2005-12-03 08:37:15.497084

(4 linhas)

É um longo caminho de INSERT ... SELECT até estes resultados, e a descrição da transformação da árvore de comando será a última neste capítulo. Primeiro existe a saída do analisador:

```
INSERT INTO tbl_cadarço_ok
SELECT tbl_cadarço_chegada.cheg_nome, tbl_cadarço_chegada.cheg_quant
FROM tbl_cadarço_chegada tbl_cadarço_chegada, tbl_cadarço_ok tbl_cadarço_ok;
```

Depois é aplicada a primeira regra a `tbl_cadarço_ok_ins` resultando em

```
UPDATE vis_cadarço
SET cad_sap_num_par_disp = vis_cadarço.cad_sap_num_par_disp +
                           tbl_cadarço_chegada.cheg_quant
FROM tbl_cadarço_chegada tbl_cadarço_chegada, tbl_cadarço_ok tbl_cadarço_ok,
tbl_cadarço_ok *OLD*, tbl_cadarço_ok *NEW*,
vis_cadarço vis_cadarço
WHERE vis_cadarço.cad_sap_nome = tbl_cadarço_chegada.cheg_nome;
```

e jogado fora o INSERT original em `tbl_cadarço_ok`. Este comando reescrito é passado para o sistema de regras novamente, e a aplicação segunda regra a `vis_cadarço_upd` produz

```
UPDATE tbl_cadarço
SET cad_sap_nome = vis_cadarço.cad_sap_nome,
    cad_sap_num_par_disp = vis_cadarço.cad_sap_num_par_disp +
tbl_cadarço_chegada.cheg_quant,
    cad_sap_cor = vis_cadarço.cad_sap_cor,
    cad_sap_comp = vis_cadarço.cad_sap_comp,
    cad_sap_unid = vis_cadarço.cad_sap_unid
```

```

FROM tbl_cadarço_chegada tbl_cadarço_chegada, tbl_cadarço_ok tbl_cadarço_ok,
     tbl_cadarço_ok *OLD*, tbl_cadarço_ok *NEW*,
     vis_cadarço vis_cadarço, vis_cadarço *OLD*,
     vis_cadarço *NEW*, tbl_cadarço tbl_cadarço
WHERE vis_cadarço.cad_sap_nome = tbl_cadarço_chegada.cheg_nome
     AND tbl_cadarço.cad_sap_nome = vis_cadarço.cad_sap_nome;

```

Novamente esta regra é do tipo `INSTEAD` e a árvore de comando anterior é jogada fora. Deve ser observado que este comando ainda utiliza a visão `vis_cadarço`, mas o sistema de regras não termina neste passo, e portanto continua e aplica a regra `_RETURN` produzindo

```

UPDATE tbl_cadarço
  SET cad_sap_nome = s.cad_sap_nome,
      cad_sap_num_par_disp = s.cad_sap_num_par_disp + tbl_cadarço_chegada.cheg_quant,
      cad_sap_cor = s.cad_sap_cor,
      cad_sap_comp = s.cad_sap_comp,
      cad_sap_unid = s.cad_sap_unid
FROM tbl_cadarço_chegada tbl_cadarço_chegada, tbl_cadarço_ok tbl_cadarço_ok,
     tbl_cadarço_ok *OLD*, tbl_cadarço_ok *NEW*,
     vis_cadarço vis_cadarço, vis_cadarço *OLD*,
     vis_cadarço *NEW*, tbl_cadarço tbl_cadarço,
     vis_cadarço *OLD*, vis_cadarço *NEW*,
     tbl_cadarço s, tbl_unidade u
WHERE s.cad_sap_nome = tbl_cadarço_chegada.cheg_nome
     AND tbl_cadarço.cad_sap_nome = s.cad_sap_nome;

```

Por fim a regra `reg_cadarço_upd` é aplicada produzindo a árvore de comando adicional

```

INSERT INTO tbl_cadarço_log
SELECT s.cad_sap_nome,
      s.cad_sap_num_par_disp + tbl_cadarço_chegada.cheg_quant,
      current_user,
      current_timestamp
FROM tbl_cadarço_chegada tbl_cadarço_chegada, tbl_cadarço_ok tbl_cadarço_ok,
     tbl_cadarço_ok *OLD*, tbl_cadarço_ok *NEW*,
     vis_cadarço vis_cadarço, vis_cadarço *OLD*,
     vis_cadarço *NEW*, tbl_cadarço tbl_cadarço,
     vis_cadarço *OLD*, vis_cadarço *NEW*,
     tbl_cadarço s, tbl_unidade u,
     tbl_cadarço *OLD*, tbl_cadarço *NEW*
     tbl_cadarço_log tbl_cadarço_log
WHERE s.cad_sap_nome = tbl_cadarço_chegada.cheg_nome
     AND tbl_cadarço.cad_sap_nome = s.cad_sap_nome
     AND (s.cad_sap_num_par_disp + tbl_cadarço_chegada.cheg_quant) <> s.cad_sap_num_par_disp;

```

Após isto o sistema de regras esgota as regras, e retorna as árvores de comando geradas.

Desta maneira se termina com duas árvores de comando finais equivalentes às instruções SQL

```

INSERT INTO tbl_cadarço_log
SELECT s.cad_sap_nome,
      s.cad_sap_num_par_disp + tbl_cadarço_chegada.cheg_quant,
      current_user,
      current_timestamp
FROM tbl_cadarço_chegada tbl_cadarço_chegada, tbl_cadarço tbl_cadarço,
     tbl_cadarço s
WHERE s.cad_sap_nome = tbl_cadarço_chegada.cheg_nome
     AND tbl_cadarço.cad_sap_nome = s.cad_sap_nome
     AND s.cad_sap_num_par_disp + tbl_cadarço_chegada.cheg_quant <> s.cad_sap_num_par_disp;

UPDATE tbl_cadarço
  SET cad_sap_num_par_disp = tbl_cadarço.cad_sap_num_par_disp +
                             tbl_cadarço_chegada.cheg_quant

```

```
FROM tbl_cadarço_chegada tbl_cadarço_chegada,
     tbl_cadarço tbl_cadarço,
     tbl_cadarço s
WHERE s.cad_sap_nome = tbl_cadarço_chegada.cad_sap_nome
      AND tbl_cadarço.cad_sap_nome = s.cad_sap_nome;
```

O resultado é que a inserção dos dados de uma relação em outra, mudados para atualizações em uma terceira, e mudados para atualizações em uma quarta mais o acompanhamento da atualização final em uma quinta, acaba reduzida a dois comandos.

Existe um pequeno detalhe um pouco feio. Olhando os dois comandos vê-se que a relação `tbl_cadarço` aparece duas vezes na tabela de abrangência, quando poderia com certeza ser reduzida para uma vez. O planejador não trata isto e, portanto, o plano de execução para a saída do sistema de regras do `INSERT` será

```
Nested Loop
-> Merge Join
    -> Seq Scan
        -> Sort
            -> Seq Scan on s
    -> Seq Scan
        -> Sort
            -> Seq Scan on tbl_cadarço_chegada
-> Seq Scan on tbl_cadarço
```

enquanto se fosse omitida a entrada adicional na tabela de abrangência seria

```
Merge Join
-> Seq Scan
    -> Sort
        -> Seq Scan on s
-> Seq Scan
    -> Sort
        -> Seq Scan on tbl_cadarço_chegada
```

que produz exatamente as mesmas entradas na tabela de acompanhamento. Portanto, o sistema de regras causa uma varredura adicional não necessária na tabela `tbl_cadarço`, e a mesma varredura redundante é feita uma vez mais no `UPDATE`, mas foi realmente um trabalho duro tornar isto tudo possível.

Agora é feita uma demonstração final do sistema de regras do PostgreSQL e de seu poder. Digamos que se deseja adicionar alguns cadarços com cores extraordinárias ao banco de dados:

```
INSERT INTO vis_cadarço VALUES ('cad9', 0, 'ciano', 35.0, 'inch', 0.0);
INSERT INTO vis_cadarço VALUES ('cad10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

Deseja-se construir uma visão para verificar quais entradas em `vis_cadarço` não correspondem a nenhuma cor de sapato. A visão para esta finalidade é

```
CREATE VIEW vis_cadarço_não_combina AS
  SELECT * FROM vis_cadarço WHERE NOT EXISTS
    (SELECT sap_nome FROM vis_sapato WHERE sap_cor_cad_pref = cad_sap_cor);
```

e sua saída é

```
SELECT * FROM vis_cadarço_não_combina;
```

cad_sap_nome	cad_sap_num_par_disp	cad_sap_cor	cad_sap_comp	cad_sap_unid	cad_sap_comp_cm
cad10	1000	magenta	40	inch	101.6
cad9	0	ciano	35	inch	88.9

(2 linhas)

Agora desejamos fazer com que os cadarços cuja cor não corresponde a nenhuma cor de sapato, e que não estão no estoque, sejam eliminados do banco de dados. Para tornar as coisas um pouco mais difícil para o PostgreSQL, a exclusão não é feita diretamente. Em vez disso, é criada mais uma visão

```
CREATE VIEW vis_cadarço_pode_excluir AS
  SELECT * FROM vis_cadarço_não_combina WHERE cad_sap_num_par_disp = 0;
```

e feito desta maneira:

```
DELETE FROM vis_cadarço WHERE EXISTS
  (SELECT * FROM vis_cadarço_pode_excluir
   WHERE cad_sap_nome = vis_cadarço.cad_sap_nome);
```

Voilà:

```
SELECT * FROM vis_cadarço;
```

cad_sap_nome	cad_sap_num_par_disp	cad_sap_cor	cad_sap_comp	cad_sap_unid	cad_sap_comp_cm
cad1	5	preto	80	cm	80
cad2	6	preto	100	cm	100
cad7	6	marrom	60	cm	60
cad4	8	preto	40	inch	101.6
cad3	10	preto	35	inch	88.9
cad8	21	marrom	40	inch	101.6
cad10	1000	magenta	40	inch	101.6
cad5	4	marrom	1	m	100
cad6	20	marrom	0.9	m	90

(9 linhas)

Um comando `DELETE` em uma visão, com uma qualificação de de subcomando que no total utiliza 4 visões aninhadas/juntadas, onde uma delas possui uma qualificação de subcomando contendo uma visão e onde são utilizadas colunas da visão calculadas, acaba reescrita em uma única árvore de comando que exclui os dados requisitados da tabela real.

Provavelmente existem poucas situações no mundo real onde uma construção deste tipo é necessária, mas faz bem saber que funciona.

## 33.4. Regras e privilégios

Devido à reescrita dos comandos pelo sistema de regras do PostgreSQL, são acessadas outras tabelas e visões além daquelas utilizadas no comando original. Quando são utilizadas regras de atualização, pode incluir o acesso de escrita na tabela.

As regras de reescrita não possuem um dono em separado. O dono da relação (tabela ou visão) é automaticamente o dono das regras de reescrita definidas para a relação. O sistema de regras do PostgreSQL altera o comportamento do sistema de controle de acesso padrão. As relações utilizadas devido às regras são verificadas com relação aos privilégios do dono da regra, e não do usuário chamando a regra. Isto significa que o usuário só precisa ter os privilégios requeridos pelas tabelas e visões explicitamente nomeadas em seus comandos.

Por exemplo: Um usuário possuindo uma lista de números de telefone onde alguns são privados e outros são de interesse da secretária do escritório, pode definir o seguinte:

```
CREATE TABLE tbl_telefone (pessoa text, telefone text, privado boolean);
CREATE VIEW vis_telefone AS
  SELECT pessoa, telefone FROM tbl_telefone WHERE NOT privado;
GRANT SELECT ON vis_telefone TO secretaria;
```

Assim, ninguém exceto ele próprio (e os superusuários do banco de dados) pode acessar a tabela `tbl_telefone`, mas por causa do `GRANT` a secretária pode executar o `SELECT` na visão `vis_telefone`. O sistema de regras reescreve o `SELECT` da visão `vis_telefone` em um `SELECT` da tabela `tbl_telefone`, e adiciona a qualificação que são desejadas apenas as linhas onde `privado` é falso. Uma vez que o usuário é o dono da visão `vis_telefone` e, portanto, o dono da regra, o



acesso de leitura para a tabela `tbl_telefone` agora é verificado com relação aos privilégios do usuário, e o comando é permitido. A verificação do privilégio para acessar a visão `vis_telefone` também é feita, mas com relação ao usuário que acessa a visão e, portanto, ninguém além do próprio usuário e a secretária pode utilizá-la.

Os privilégios são verificados regra por regra. Portanto, agora só a secretária pode ver os números de telefone públicos. Mas a secretária pode definir uma outra visão, e permitir o acesso público a mesma. Assim, todo mundo vai poder ver os dados de `vis_telefone` através da visão da secretária. O que a secretária não pode fazer é criar uma visão com acesso direto à tabela `tbl_telefone` (Na verdade pode, mas não funciona porque todos os acessos serão negados durante a verificação de permissão). Tão logo o usuário perceba que a secretária abriu sua visão `vis_telefone` ele pode revogar o acesso. Imediatamente todos os acessos à visão da secretária vão falhar.

Pode-se pensar que esta verificação regra por regra é um furo de segurança, mas na verdade não é. Se não funcionasse desta maneira a secretária poderia definir uma tabela com as mesmas colunas de `vis_telefone`, e copiar os dados para esta tabela uma vez por dia. Então seriam seus próprios dados, e a secretária poderia conceder acesso a todos que desejasse. Um comando `GRANT` significa “Eu confio em você”. Se alguém em que você confia faz uma coisa desta, é hora de repensar e utilizar o comando `REVOKE`.

Este mecanismo também funciona para as regras de atualização. Nos exemplos da seção anterior, o dono das tabelas no banco de dados de exemplo poderia conceder para outro usuário os privilégios `SELECT`, `INSERT`, `UPDATE` e `DELETE` na visão `vis_cadarço`, e somente o privilégio `SELECT` na tabela `tbl_cadarço_log`. A ação da regra para escrever registros de acompanhamento ainda será executada com sucesso, e o outro usuário vai poder ver os registros de acompanhamento, mas se alguém criar registros falsos este não vai poder manipular nem remover estes registros falsos.

### 33.5. Regras e status dos comandos

O servidor PostgreSQL retorna uma cadeia de caracteres contendo o status do comando, como `INSERT 149592 1`, para cada comando recebido. Isto é bem simples quando não há regras envolvidas, mas o que acontece quando o comando é reescrito pelas regras?

As regras afetam o status do comando da seguinte maneira:

- Se não houver uma regra `INSTEAD` incondicional, então o comando original fornecido é executado, e o status deste comando é retornado da forma usual (mas deve ser observado que havendo regras `INSTEAD` condicionais, teria sido adicionado ao comando original a negação de suas qualificações. Isto pode reduzir o número de linhas processadas e, se acontecer, o status relatado é afetado).
- Se existir alguma regra `INSTEAD` incondicional, então o comando original não será executado. Neste caso, o servidor retorna o status do último comando inserido por uma regra `INSTEAD` (condicional ou incondicional), que é do mesmo tipo (`INSERT`, `UPDATE` ou `DELETE`) do comando original. Se não for adicionado um comando com estas características por alguma regra, então o status retornado mostra o tipo do comando original, e zero para os campos contador de linhas e OID.

(Este sistema foi estabelecido no PostgreSQL 7.3. Nas versões anteriores a esta o status do comando pode mostrar resultados diferentes quando existe uma regra.)

No segundo caso, o programador pode garantir que uma determinada regra `INSTEAD` desejada será a que vai definir o status do comando, dando a esta regra o último nome de regra na ordem alfabética entre as regras ativas, para que seja aplicada por último.

### 33.6. Regras versus gatilhos

Várias coisas que podem ser feitas utilizando gatilho também podem ser implementadas utilizando o sistema de regras do PostgreSQL. Entre o que não pode ser implementado pelas regras estão alguns tipos de restrição, especialmente as chaves estrangeiras. É possível definir uma regra qualificada para reescrever o comando como `NOTHING` se o valor da coluna não constar da outra tabela, mas neste caso os dados são desprezados em silêncio, e esta não é uma boa idéia. Se a verificação dos valores válidos for requerida, e for necessária uma mensagem de erro no caso de um valor inválido, deve ser definido através de um gatilho.

Por outro lado, um gatilho disparado pelo `INSERT` em uma visão pode fazer o mesmo que uma regra faz: colocar os dados em algum outro lugar, e suprimir a inserção na visão. Porém, para `UPDATE` e `DELETE` o gatilho não pode fazer o mesmo que

a regra faz, porque não existem dados reais na relação da visão para serem varridos e, portanto, o gatilho nunca vai ser chamado. Nestes casos só a regra funciona.

Entre o que pode ser implementado por ambos, qual é melhor depende da utilização do banco de dados. O gatilho é disparado uma vez para cada linha afetada. A regra manipula o comando, ou gera um comando adicional. Portanto, se uma instrução afetar muitas linhas é provável que a regra emitindo um comando adicional seja mais rápida que o gatilho chamado para todas as linhas, ocasionando a execução de suas operações muitas vezes. Entretanto, a abordagem do gatilho é conceitualmente bem mais simples que a abordagem da regra, sendo mais fácil para os usuários inexperientes definir um gatilho corretamente.

Abaixo está mostrado um exemplo de como a escolha entre regra e gatilho se apresenta em uma situação. Existem duas tabelas:

```
CREATE TABLE computador (
    hospedeiro    text,      -- indexado
    fabricante    text      -- indexado
);

CREATE TABLE programa (
    programa      text,      -- indexado
    hospedeiro    text      -- indexado
);
```

As duas tabelas possuem milhares de linhas, e os índices para `hospedeiro` são únicos. A regra e o gatilho devem implementar uma restrição para excluir as linhas de `programa` que fazem referência a um computador excluído. O gatilho utiliza este comando:

```
DELETE FROM programa WHERE hospedeiro = $1;
```

Uma vez que o gatilho é chamado para cada linha excluída da tabela `computador`, pode preparar e salvar o plano para este comando e passar o valor do `hospedeiro` para o parâmetro. A regra é escrita como:

```
CREATE RULE computador_del AS ON DELETE TO computador
DO DELETE FROM programa WHERE hospedeiro = OLD.hospedeiro;
```

A seguir são analisados tipos diferentes de exclusão. No caso de

```
DELETE FROM computador WHERE hospedeiro = 'meupc.local.net';
```

a tabela `computador` é varrida pelo índice (rápido), e o comando emitido pelo gatilho também utiliza uma varredura de índice (também rápido). O comando adicionado pela regra é:

```
DELETE FROM programa WHERE computador.hospedeiro = 'meupc.local.net'
AND programa.hospedeiro = computador.hospedeiro;
```

Uma vez que existem índices apropriados definidos, o planejador cria o plano

```
Nestloop
-> Index Scan using comp_hospidx on computador
-> Index Scan using prog_hospidx on programa
```

Portanto, não há muita diferença de velocidade entre a implementação do gatilho e da regra.

Na próxima exclusão serão eliminados 2000 computadores onde `hospedeiro` começa por `old`. Existem dois comandos possíveis de serem utilizados. Um é

```
DELETE FROM computador WHERE hospedeiro >= 'old'
AND hospedeiro < 'ole'
```

O comando adicionado pela regra é

```
DELETE FROM programa WHERE computador.hospedeiro >= 'old' AND computador.hospedeiro < 'ole'
      AND programa.hospedeiro = computador.hospedeiro;
```

com o plano

```
Hash Join
-> Seq Scan on programa
-> Hash
    -> Index Scan using comp_hospidx on computador
```

O outro comando possível é

```
DELETE FROM computador WHERE hospedeiro ~ '^old';
```

que resulta no seguinte plano de execução para o comando adicionado pela regra:

```
Nestloop
-> Index Scan using comp_hospidx on computador
-> Index Scan using prog_hospidx on programa
```

Isto mostra que o planejador não percebe que a qualificação para hospedeiro em computador também pode ser utilizada para uma varredura de índice em programa quando há expressões de qualificação combinadas por AND, que é o que o planejador faz na versão do comando com expressão regular. O gatilho é chamado uma vez para cada um dos 2000 computadores antigos a serem excluídos, e isto resulta em uma varredura de índice para computador e 2000 varreduras de índice para programa. A implementação da regra faz isto com dois comandos que utilizam índices. Se a regra é mais rápida que a situação da varredura seqüencial, depende do tamanho total da tabela programa. A execução de 2000 comandos pelo gatilho através do gerenciador da SPI toma algum tempo, mesmo que todos os blocos do índice fiquem rapidamente no cache.

O último comando a ser visto é

```
DELETE FROM computador WHERE fabricante = 'bim';
```

Novamente poderia resultar em muitas linhas sendo excluídas de computador. Portanto, o gatilho novamente executa muitos comandos através do executor. O comando gerado pela regra é

```
DELETE FROM programa WHERE computador.fabricante = 'bim'
      AND programa.hospedeiro = computador.hospedeiro;
```

O plano para este comando novamente é um laço aninhado sobre duas varreduras de índice, apenas usando um índice diferente para computador:

```
Nestloop
-> Index Scan using comp_fabridx on computador
-> Index Scan using prog_hospidx on programa
```

Em qualquer um destes casos, os comandos adicionais do sistema de regras são mais ou menos independentes do número de linhas afetadas pelo comando.

Em resumo, as regras somente são significativamente mais lentas que os gatilhos quando suas ações resultam em junções grandes e mal qualificadas, uma situação onde o planejador falha.

# Capítulo 34. Linguagens procedurais

O PostgreSQL permite que as funções definidas pelo usuário sejam escritas em outras linguagens além de SQL e C. Estas linguagens são chamadas genericamente de *linguagens procedurais* (PLs). No caso de uma função escrita em uma linguagem procedural, o servidor de banco de dados não possui nenhum conhecimento interno sobre como interpretar o texto do código fonte da função. Em vez disso, a tarefa é passada para um tratador especial que conhece os detalhes da linguagem. O tratador pode fazer todo o trabalho de análise gramatical e sintática, execução, etc., por si próprio, ou pode servir como um “elo de ligação” entre o PostgreSQL e a implementação existente de uma linguagem de programação. O tratador em si é uma função escrita na linguagem C, compilada como um objeto compartilhado, e carregado conforme necessário, como qualquer outra função escrita na linguagem C.

Atualmente existem quatro linguagens procedurais disponíveis na distribuição padrão PostgreSQL: PL/pgSQL (Capítulo 35), PL/Tcl (Capítulo 36), PL/Perl (Capítulo 37) e PL/Python (Capítulo 38). Os usuários podem definir outras linguagens. Os princípios básicos para o desenvolvimento de uma nova linguagem procedural estão descritos no Capítulo 45.

Estão disponíveis outras linguagens procedurais adicionais, mas não são incluídas na distribuição núcleo. O Apêndice H contém informações sobre como encontrá-las.

## 34.1. Instalação de linguagem procedural

A linguagem procedural deve ser “instalada” em cada banco de dados onde vai ser utilizada. Porém, as linguagens procedurais instaladas no banco de dados `template1` ficam disponíveis automaticamente em todos os bancos de dados criados após sua instalação, uma vez que suas entradas em `template1` são copiadas pelo comando `CREATE DATABASE`. Portanto, o administrador de banco de dados pode decidir quais linguagens ficarão disponíveis em quais bancos de dados, e pode tornar algumas linguagens disponíveis por padrão se assim o decidir.

Para as linguagens fornecidas na distribuição padrão, pode ser utilizado o programa `createlang` para instalar a linguagem em vez de executar os passos manualmente. Por exemplo, para instalar a linguagem PL/pgSQL no banco de dados `template1` é utilizado:

```
createlang plpgsql template1
```

O procedimento manual descrito abaixo somente é recomendado para a instalação de linguagens personalizadas que o programa `createlang` desconhece.

### Instalação manual de linguagem procedural

A linguagem procedural é instalada no banco de dados em quatro passos, que devem ser efetuados por um superusuário do banco de dados. O programa `createlang` automatiza tudo menos o passo 1.

1. O objeto compartilhado contendo o tratador da linguagem deve ser compilado e instalado em um diretório de biblioteca apropriado. Funciona da mesma maneira que a construção e instalação de módulos contendo funções C regulares definidas pelo usuário; consulte a Seção 31.9.6. Geralmente o tratador da linguagem depende de uma biblioteca externa que disponibiliza o mecanismo verdadeiro da linguagem de programação; se for assim, esta biblioteca também deve ser instalada.
2. O tratador deve ser declarado pelo comando:

```
CREATE FUNCTION nome_da_função_tratadora()  
  RETURNS language_handler  
  AS 'caminho_para_o_objeto_compartilhado'  
  LANGUAGE C;
```

O tipo especial retornado `language_handler` informa ao sistema de banco de dados que a função não retorna um dos tipos de dado SQL definidos, e que não pode ser utilizada diretamente nas declarações SQL.

3. Opcionalmente o tratador da linguagem pode disponibilizar uma função “validadora” para verificar se a definição da função está correta, sem na verdade executá-la. Caso exista, a função validadora é chamada pelo comando `CREATE FUNCTION`. Se o tratador disponibilizar uma função validadora, esta deve ser declarada por um comando como:

```
CREATE FUNCTION nome_da_função_validadora(oid)
```

```

RETURNS void
AS 'caminho_para_o_objeto_compartilhado'
LANGUAGE C;

```

4. A linguagem procedural deve ser declarada pelo comando:

```

CREATE [TRUSTED] [PROCEDURAL] LANGUAGE nome_da_linguagem
HANDLER nome_da_função_tratadora
[VALIDATOR nome_da_função_validadora] ;

```

A palavra opcional **TRUSTED** (confiável) especifica que é permitido aos usuários comuns do banco de dados, que não possuem privilégio de superusuário, utilizarem esta linguagem para criar procedimentos de funções e gatilhos. Uma vez que as funções na linguagem procedural são executadas dentro do servidor de banco de dados, o sinalizador **TRUSTED** somente deve ser especificado para as linguagens que não permitem acesso às funcionalidades internas do servidor de banco de dados, nem ao sistema de arquivos. As linguagens PL/pgSQL, PL/Tcl e PL/Perl são consideradas confiáveis; as linguagens PL/TclU, PL/PerlU e PL/PythonU foram projetadas para fornecer funcionalidades ilimitadas, *não* devendo ser marcadas como confiáveis.

O Exemplo 34-1 mostra como funciona o procedimento de instalação manual com a linguagem PL/pgSQL.

#### Exemplo 34-1. Instalação manual do PL/pgSQL

O comando abaixo informa ao servidor de banco de dados onde encontrar o objeto compartilhado da função tratadora de chamadas da linguagem PL/pgSQL:

```

CREATE FUNCTION plpgsql_call_handler() RETURNS language_handler AS
'$libdir/plpgsql' LANGUAGE C;

```

A linguagem PL/pgSQL possui uma função validadora, portanto esta também é declarada:

```

CREATE FUNCTION plpgsql_validator(oid) RETURNS void AS
'$libdir/plpgsql' LANGUAGE C;

```

Depois o comando

```

CREATE TRUSTED PROCEDURAL LANGUAGE plpgsql
HANDLER plpgsql_call_handler
VALIDATOR plpgsql_validator;

```

define que a função declarada anteriormente deve ser chamada para os procedimentos de função e gatilho onde o atributo de linguagem for **plpgsql**.

Na instalação padrão do PostgreSQL o tratador para a linguagem PL/pgSQL é construído e instalado no diretório de “biblioteca”. Se o suporte à linguagem Tcl estiver configurado, os tratadores para PL/Tcl e PL/TclU também serão construídos e instalados no mesmo local. Da mesma maneira, os tratadores para PL/Perl e PL/PerlU serão construídos e instalados se o suporte à linguagem Perl estiver configurado, e PL/PythonU será instalado se o suporte à linguagem Python estiver configurado.

# Capítulo 35. PL/pgSQL - Linguagem procedural SQL

PL/pgSQL é uma linguagem procedural carregável desenvolvida para o sistema de banco de dados PostgreSQL. Os objetivos de projeto da linguagem PL/pgSQL foram no sentido de criar uma linguagem procedural carregável que pudesse:

- ser utilizada para criar procedimentos de funções e de gatilhos;
- adicionar estruturas de controle à linguagem SQL;
- realizar processamentos complexos;
- herdar todos os tipos de dado, funções e operadores definidos pelo usuário;
- ser definida como confiável pelo servidor;
- ser fácil de utilizar.

Exceto pelas funções de conversão de entrada/saída e cálculos para os tipos definidos pelo usuário, tudo mais que pode ser definido por uma função escrita na linguagem C também pode ser feito usando PL/pgSQL. Por exemplo, é possível criar funções para cálculos condicionais complexos e depois usá-las para definir operadores ou em expressões de índice.

## 35.1. Visão geral

O tratador de chamadas da linguagem PL/pgSQL analisa o texto do código fonte da função e produz uma árvore de instruções binária interna, na primeira vez em que a função é chamada (em cada sessão). A árvore de instruções traduz inteiramente a estrutura da declaração PL/pgSQL, mas as expressões SQL individuais e os comandos SQL utilizados na função não são traduzidos imediatamente.

Assim que cada expressão ou comando SQL é utilizado pela primeira vez na função, o interpretador do PL/pgSQL cria um plano de execução preparado (utilizando as funções `SPI_prepare` e `SPI_saveplan` do gerenciador da Interface de Programação do Servidor - SPI). As execuções posteriores da expressão ou do comando reutilizam o plano preparado. Por isso, uma função com código condicional, contendo muitas declarações que podem requerer um plano de execução, somente prepara e salva os planos realmente utilizados durante o espaço de tempo da conexão com o banco de dados. Isto pode reduzir muito a quantidade total de tempo necessário para analisar e gerar os planos de execução para as declarações na função PL/pgSQL. A desvantagem é que erros em uma determinada expressão ou comando podem não ser detectados até que a parte da função onde se encontram seja executada.

Uma vez que o PL/pgSQL tenha construído um plano de execução para um determinado comando da função, este plano será reutilizado enquanto durar a conexão com o banco de dados. Normalmente há um ganho de desempenho, mas pode causar problema se o esquema do banco de dados for modificado dinamicamente. Por exemplo:

```
CREATE FUNCTION populate() RETURNS integer AS $$
DECLARE
    -- declarações
BEGIN
    PERFORM minha_funcao();
END;
$$ LANGUAGE plpgsql;
```

Se a função acima for executada fará referência ao OID da `minha_funcao()` no plano de execução gerado para a instrução `PERFORM`. Mais tarde, se a função `minha_funcao()` for removida e recriada, então `populate()` não vai mais conseguir encontrar `minha_funcao()`. Por isso é necessário recriar `populate()`, ou pelo menos começar uma nova sessão de banco de dados para que a função seja compilada novamente. Outra forma de evitar este problema é utilizar `CREATE OR REPLACE FUNCTION` ao atualizar a definição de `minha_funcao` (quando a função é “substituída” o OID não muda).

Uma vez que o PL/pgSQL salva os planos de execução desta maneira, os comandos SQL que aparecem diretamente na função PL/pgSQL devem fazer referência às mesmas tabelas e colunas em todas as execuções; ou seja, não pode ser utilizado um parâmetro como nome de tabela ou de coluna no comando SQL. Para contornar esta restrição podem ser construídos comandos dinâmicos utilizando a instrução `EXECUTE` do PL/pgSQL — o preço a ser pago é a construção de um novo plano de execução a cada execução.

**Nota:** A instrução `EXECUTE` do PL/pgSQL não tem relação com a instrução `EXECUTE` do SQL suportada pelo servidor PostgreSQL. A instrução `EXECUTE` do servidor não pode ser utilizada dentro das funções PL/pgSQL (e não é necessário).

### 35.1.1. Vantagens da utilização da linguagem PL/pgSQL

A linguagem SQL é a que o PostgreSQL (e a maioria dos bancos de dados relacionais) utiliza como linguagem de comandos. É portátil e fácil de ser aprendida. Entretanto, todas as declarações SQL devem ser executadas individualmente pelo servidor de banco de dados.

Isto significa que o aplicativo cliente deve enviar o comando para o servidor de banco de dados, aguardar que seja processado, receber os resultados, realizar algum processamento, e enviar o próximo comando para o servidor. Tudo isto envolve comunicação entre processos e pode, também, envolver tráfego na rede se o cliente não estiver na mesma máquina onde se encontra o servidor de banco de dados.

Usando a linguagem PL/pgSQL pode ser agrupado um bloco de processamento e uma série de comandos *dentro* do servidor de banco de dados, juntando o poder da linguagem procedural com a facilidade de uso da linguagem SQL, e economizando muito tempo, porque não há necessidade da sobrecarga de comunicação entre o cliente e o servidor. Isto pode aumentar o desempenho consideravelmente.

Também podem ser utilizados na linguagem PL/pgSQL todos os tipos de dados, operadores e funções da linguagem SQL.

### 35.1.2. Tipos de dado suportados nos argumentos e no resultado

As funções escritas em PL/pgSQL aceitam como argumento qualquer tipo de dado escalar ou matriz suportado pelo servidor, e podem retornar como resultado qualquer um destes tipos. As funções também aceitam e retornam qualquer tipo composto (tipo linha) especificado por nome. Também é possível declarar uma função PL/pgSQL como retornando `record`, significando que o resultado é um tipo linha, cujas colunas são determinadas pela especificação no comando que faz a chamada, conforme mostrado na Seção 7.2.1.4.

As funções PL/pgSQL também podem ser declaradas como recebendo ou retornando os tipos polimórficos `anyelement` e `anyarray`. Os tipos de dado verdadeiros tratados pelas funções polimórficas podem variar entre chamadas, conforme mostrado na Seção 31.2.5. Na Seção 35.4.1 é mostrado um exemplo.

As funções PL/pgSQL também podem ser declaradas como retornando “set” (conjunto), ou tabela, de qualquer tipo de dado para o qual pode ser retornada uma única instância. Este tipo de função gera sua saída executando `RETURN NEXT` para cada elemento desejado do conjunto resultado.

Por fim, uma função PL/pgSQL pode ser declarada como retornando `void` se não produzir nenhum valor de retorno útil.

Atualmente a linguagem PL/pgSQL não possui suporte total para os tipos domínio: trata o domínio da mesma maneira que o tipo escalar subjacente. Isto significa que não obriga respeitar as restrições associadas ao domínio, o que não representa problema para os argumentos da função, mas é perigoso declarar uma função PL/pgSQL como retornando um tipo domínio.

## 35.2. Dicas para desenvolvimento em PL/pgSQL

Uma boa maneira de desenvolver em PL/pgSQL é utilizar o editor de texto preferido para criar as funções e, em outra janela, utilizar o `psql` para carregar e testar as funções desenvolvidas. Se estiver sendo feito desta maneira, é uma boa idéia escrever a função utilizando `CREATE OR REPLACE FUNCTION`. Fazendo assim, basta recarregar o arquivo para atualizar a definição da função. Por exemplo:

```
CREATE OR REPLACE FUNCTION funcao_teste(integer) RETURNS integer AS $$
    ....
$$ LANGUAGE plpgsql;
```

Na linha de comando do `psql` a definição da função pode ser carregada ou recarregada utilizando

```
\i nome_do_arquivo.sql
```

e logo em seguida podem ser executados os comandos SQL que testam a função.

Outra boa maneira de desenvolver em PL/pgSQL, é utilizar uma ferramenta de acesso ao banco de dados com interface gráfica que facilite o desenvolvimento em linguagem procedural. Um exemplo deste tipo de ferramenta é o PgAccess, mas

existem outras. Estas ferramentas geralmente disponibilizam funcionalidades úteis como o escape de apóstrofes, e tornam mais fácil recriar e depurar funções.

### 35.2.1. Tratamento dos apóstrofes

O código da função PL/pgSQL é especificado no comando `CREATE FUNCTION` como um literal cadeia de caracteres. Se o literal cadeia de caracteres for escrito da maneira usual, que é entre apóstrofes (`'`), então os apóstrofes dentro do corpo da função devem ser duplicados; da mesma maneira, as contrabarras dentro do corpo da função (`\`) devem ser duplicadas. Duplicar os apóstrofes é no mínimo entediante, e nos casos mais complicados pode tornar o código difícil de ser compreendido, porque pode-se chegar facilmente a uma situação onde são necessários seis ou mais apóstrofes adjacentes. Por isso, recomenda-se que o corpo da função seja escrito em um literal cadeia de caracteres delimitado por “cifrão” (consulte a Seção 4.1.2.2) em vez de delimitado por apóstrofes. Na abordagem delimitada por cifrão os apóstrofes nunca são duplicados e, em vez disso, toma-se o cuidado de escolher uma marca diferente para cada nível de aninhamento necessário. Por exemplo, o comando `CREATE FUNCTION` pode ser escrito da seguinte maneira:

```
CREATE OR REPLACE FUNCTION funcao_teste(integer) RETURNS integer AS $PROC$
    ....
$PROC$ LANGUAGE plpgsql;
```

No corpo da função podem ser utilizados apóstrofes para delimitar cadeias de caracteres simples nos comandos SQL, e `$$` para delimitar fragmentos de comandos SQL montados como cadeia de caracteres. Se for necessário delimitar um texto contendo `$$`, deve ser utilizado `$Q$`, e assim por diante.

O quadro abaixo mostra o que deve ser feito para escrever o corpo da função entre apóstrofes (sem uso da delimitação por cifrão). Pode ser útil para tornar códigos anteriores à delimitação por cifrão mais fácil de serem compreendidos.

#### 1 apóstrofo

para começar e terminar o corpo da função como, por exemplo:

```
CREATE FUNCTION foo() RETURNS integer AS '
    ....
' LANGUAGE plpgsql;
```

Em todas as ocorrências dentro do corpo da função os apóstrofes *devem* aparecer em pares.

#### 2 apóstrofes

Para literais cadeia de caracteres dentro do corpo da função como, por exemplo:

```
a_output := 'Blah';
SELECT * FROM users WHERE f_nome='foobar';
```

Na abordagem delimitada por cifrão seria escrito apenas

```
a_output := 'Blah';
SELECT * FROM users WHERE f_nome='foobar';
```

que é exatamente o código visto pelo analisador do PL/pgSQL nos dois casos.

#### 4 apóstrofes

Quando é necessário colocar um apóstrofo em uma constante cadeia de caracteres dentro do corpo da função como, por exemplo:

```
a_output := a_output || ' AND nome LIKE '''foobar''' AND xyz'
```

O verdadeiro valor anexado a `a_output` seria: `AND nome LIKE 'foobar' AND xyz`.

Na abordagem delimitada por cifrão seria escrito

```
a_output := a_output || $$ AND nome LIKE 'foobar' AND xyz$$
```

tendo-se o cuidado de que todos os delimitadores por cifrão envolvendo este comando não sejam apenas `$$`.

#### 6 apóstrofes

Quando o apóstrofo na cadeia de caracteres dentro do corpo da função está adjacente ao final da constante cadeia de caracteres como, por exemplo:

```
a_output := a_output || ' AND nome LIKE '''foobar''''
```



O valor anexado à a\_output seria: AND nome LIKE 'foobar'.

Na abordagem delimitada por cifrão se tornaria

```
a_output := a_output || $$ AND nome LIKE 'foobar'$$
```

10 apóstrofos

Quando é necessário colocar dois apóstrofes em uma constante cadeia de caracteres (que necessita de 8 apóstrofes), e estes dois apóstrofes estão adjacentes ao final da constante cadeia de caracteres (mais 2 apóstrofes). Normalmente isto só é necessário quando são escritas funções que geram outras funções como no Exemplo 35-8. Por exemplo:

```
a_output := a_output || ' if v_' ||  
referrer_keys.kind || ' like ''' ||  
|| referrer_keys.key_string || '''' ||  
then return ''' || referrer_keys.referrer_type  
|| '''; end if;'''
```

O valor de `a_output` seria então:

```
if v_... like '...' then return '...'; end if;
```

Na abordagem delimitada por cifrão se tornaria

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
|| referrer_keys.key_string || $$'
then return '$$ || referrer_keys.referrer_type
|| $$'; end if;$$;
```

onde se assume que só é necessário colocar um único apóstrofo em `a_output`, porque este será delimitado novamente antes de ser utilizado.

Uma outra abordagem é fazer o escape dos apóstrofes no corpo da função utilizando a contrabarra em vez de duplicá-los. Desta forma é escrito `\'` no lugar de `'`. Alguns acham esta forma mais fácil, porém outros não concordam.

### 35.3. Estrutura da linguagem PL/pgSQL

A linguagem PL/pgSQL é estruturada em blocos. O texto completo da definição da função deve ser um *bloco*. Um bloco é definido como:

```
[ <<rótulo>> ]
[ DECLARE
    declarações ]
BEGIN
    instruções
END;
```

Todas as declarações e instruções dentro do bloco devem ser terminadas por ponto-e-vírgula. Um bloco contido dentro de outro bloco deve conter um ponto-e-vírgula após o `END`, conforme mostrado acima; entretanto, o `END` final que conclui o corpo da função não requer o ponto-e-vírgula.

Todas as palavras chave e identificadores podem ser escritos misturando letras maiúsculas e minúsculas. As letras dos identificadores são convertidas implicitamente em minúsculas, a menos que estejam entre aspas.

Existem dois tipos de comentários no PL/pgSQL. O hífen duplo (--) começa um comentário que se estende até o final da linha. O /\* começa um bloco de comentário que se estende até a próxima ocorrência de \*/. Os blocos de comentário não podem ser aninhados, mas comentários de hífen duplo podem estar contidos em blocos de comentário, e os hífens duplos escondem os delimitadores de bloco de comentário /\* e \*/.

Qualquer instrução na seção de instruções do bloco pode ser um *sub-bloco*. Os sub-blocos podem ser utilizados para agrupamento lógico, ou para limitar o escopo de variáveis a um pequeno grupo de instruções.

As variáveis declaradas na seção de declaração que precede um bloco são inicializadas com seu valor padrão toda vez que o bloco é executado, e não somente uma vez a cada chamada da função. Por exemplo:

```
CREATE FUNCTION func_escopo() RETURNS integer AS $$
DECLARE
    quantidade integer := 30;
```

```

BEGIN
    RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 30
    quantidade := 50;
    --
    -- Criar um sub-bloco
    --
    DECLARE
        quantidade integer := 80;
    BEGIN
        RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 80
    END;

    RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 50

    RETURN quantidade;
END;
$$ LANGUAGE plpgsql;

=> SELECT func_escopo();

```

```

NOTA: Aqui a quantidade é 30
NOTA: Aqui a quantidade é 80
NOTA: Aqui a quantidade é 50
func_escopo
-----
          50
(1 linha)

```

É importante não confundir a utilização de `BEGIN/END` para o agrupamento de instruções na linguagem PL/pgSQL, com os comandos de banco de dados para controle de transação. O `BEGIN/END` da linguagem PL/pgSQL é apenas para agrupamento; não começam nem terminam transações. Os procedimentos de funções e de gatilhos são sempre executados dentro da transação estabelecida pelo comando externo — não podem iniciar ou efetivar transações, porque não haveria contexto para estas serem executadas. Entretanto, um bloco contendo a cláusula `EXCEPTION` forma, efetivamente, uma subtransação que pode ser desfeita sem afetar a transação externa. Para obter mais detalhes deve ser consultada a Seção 35.7.5.

## 35.4. Declarações

Todas as variáveis utilizadas em um bloco devem ser declaradas na seção de declarações do bloco (A única exceção é a variável de laço do `FOR` interagindo sobre um intervalo de valores inteiros, que é automaticamente declarada como sendo do tipo inteiro).

As variáveis da linguagem PL/pgSQL podem possuir qualquer tipo de dado da linguagem SQL, como `integer`, `varchar` e `char`.

Abaixo seguem alguns exemplos de declaração de variáveis:

```

id_usuario    integer;
quantidade    numeric(5);
url           varchar;
minha_linha   nome_da_tabela%ROWTYPE;
meu_campo     nome_da_tabela.nome_da_coluna%TYPE;
uma_linha     RECORD;

```

A sintaxe geral para declaração de variáveis é:

```
nome [ CONSTANT ] tipo [ NOT NULL ] [ { DEFAULT | := } expressão ];
```

A cláusula `DEFAULT`, se for fornecida, especifica o valor inicial atribuído à variável quando o processamento entra no bloco. Se a cláusula `DEFAULT` não for fornecida, então a variável é inicializada com o valor nulo do SQL. A opção `CONSTANT` impede que seja atribuído valor a variável e, portanto, seu valor permanece constante pela duração do bloco. Se

for especificado NOT NULL, uma atribuição de valor nulo resulta em um erro em tempo de execução. Todas as variáveis declaradas como NOT NULL devem ter um valor padrão não nulo especificado.

O valor padrão é avaliado toda vez que a execução entra no bloco. Portanto, por exemplo, atribuir `now()` a uma variável do tipo `timestamp` faz com que a variável possua a data e hora da chamada corrente à função, e não de quando a função foi pré-compilada.

Exemplos:

```
quantidade integer DEFAULT 32;
url          varchar := 'http://meu-site.com';
id_usuario   CONSTANT integer := 10;
```

### 35.4.1. Aliases para parâmetros de função

Os parâmetros passados para as funções recebem como nome os identificadores `$1`, `$2`, etc. Opcionalmente, para melhorar a legibilidade do código, podem ser declarados aliases para os nomes dos parâmetros `$n`. Para fazer referência ao valor do parâmetro, pode ser utilizado tanto o aliás quanto o identificador numérico.

Existem duas maneiras de criar um aliás. A forma preferida é fornecer nome ao parâmetro no comando `CREATE FUNCTION` como, por exemplo:

```
CREATE FUNCTION taxa_de_venda(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

A outra maneira, que era a única disponível antes da versão 8.0 do PostgreSQL, é declarar explicitamente um aliás utilizando a sintaxe de declaração

```
nome ALIAS FOR $n;
```

O exemplo acima escrito utilizando este estilo fica da seguinte maneira:

```
CREATE FUNCTION taxa_de_venda(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Alguns outros exemplos:

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index    ALIAS FOR $2;
BEGIN
    -- algum processamento neste ponto
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION concatenar_campos_selecionados(in_t nome_da_tabela) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

Quando o tipo retornado por uma função PL/pgSQL é declarado como sendo de um tipo polimórfico (`anyelement` ou `anyarray`), é criado o parâmetro especial `$0`. Seu tipo de dado é o tipo de dado real a ser retornado pela função, conforme deduzido a partir dos tipos da entrada corrente (consulte a Seção 31.2.5). Isto permite à função descobrir o verdadeiro tipo de dado retornado para a entrada corrente conforme mostrado na Seção 35.4.2. O parâmetro `$0` é inicializado como nulo e pode ser modificado pela função, portanto pode ser utilizado para armazenar o valor a ser retornado se for desejado, embora não seja requerido. Também pode ser criado um aliás para o parâmetro `$0`. Por exemplo, esta função funciona com qualquer tipo de dado que possua o operador `+`:

```
CREATE FUNCTION somar_tres_valores(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    resultado ALIAS FOR $0;
BEGIN
    resultado := v1 + v2 + v3;
    RETURN resultado;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT somar_tres_valores(10,20,30);
```

```
somar_tres_valores
-----
                60
(1 linha)
```

```
SELECT somar_tres_valores(1.1,2.2,3.3);
```

```
somar_tres_valores
-----
             6.6
(1 linha)
```

### 35.4.2. Cópia de tipo

```
variável%TYPE
```

A expressão `%TYPE` fornece o tipo de dado da variável ou da coluna da tabela. Pode ser utilizada para declarar variáveis que armazenam valores do banco de dados. Por exemplo, supondo que exista uma coluna chamada `id_usuario` na tabela `usuarios`, para declarar uma variável com o mesmo tipo de dado de `usuarios.id_usuario` deve ser escrito:

```
id_usuario usuarios.id_usuario%TYPE;
```

Utilizando `%TYPE` não é necessário conhecer o tipo de dado da estrutura sendo referenciada e, ainda mais importante, se o tipo de dado do item referenciado mudar no futuro (por exemplo: o tipo de dado de `id_usuario` for mudado de `integer` para `real`), não será necessário mudar a definição na função.

A expressão `%TYPE` é particularmente útil em funções polimórficas, uma vez que os tipos de dado das variáveis internas podem mudar de uma chamada para outra. Pode-se criar variáveis apropriadas aplicando `%TYPE` aos argumentos da função ou resultado.

### 35.4.3. Tipos linha

```
nome nome_da_tabela%ROWTYPE;
nome nome_do_tipo_composto;
```

Uma variável de tipo composto é chamada de variável *linha* (ou variável *tipo-linha*). Este tipo de variável pode armazenar toda uma linha de resultado de um comando `SELECT` ou `FOR`, desde que o conjunto de colunas do comando corresponda ao tipo declarado para a variável. Os campos individuais do valor linha são acessados utilizando a notação usual de ponto como, por exemplo, `variavel_linha.campo`.

Uma variável-linha pode ser declarada como tendo o mesmo tipo de dado das linhas de uma tabela ou de uma visão existente, utilizando a notação `nome_da_tabela%ROWTYPE`; ou pode ser declarada especificando o nome de um tipo

composto (Uma vez que todas as tabelas possuem um tipo composto associado, que possui o mesmo nome da tabela, na verdade não faz diferença para o PostgreSQL se %ROWTYPE é escrito ou não, mas a forma contendo %ROWTYPE é mais portátil).

Os parâmetros das funções podem ser de tipo composto (linhas completas da tabela). Neste caso, o identificador correspondente \$n será uma variável linha, e os campos poderão ser selecionados a partir deste identificador como, por exemplo, \$1.id\_usuario.

Somente podem ser acessadas na variável tipo-linha as colunas definidas pelo usuário presentes na linha da tabela, a coluna OID e as outras colunas do sistema não podem ser acessadas por esta variável (porque a linha pode ser de uma visão). Os campos do tipo-linha herdam o tamanho do campo da tabela, ou a precisão no caso de tipos de dado como char(n).

Abaixo está mostrado um exemplo de utilização de tipo composto:

```
CREATE FUNCTION mesclar_campos(t_linha nome_da_tabela) RETURNS text AS $$
DECLARE
    t2_linha nome_tabela2%ROWTYPE;
BEGIN
    SELECT * INTO t2_linha FROM nome_tabela2 WHERE ... ;
    RETURN t_linha.f1 || t2_linha.f3 || t_linha.f5 || t2_linha.f7;
END;
$$ LANGUAGE plpgsql;

SELECT mesclar_campos(t.*) FROM nome_da_tabela t WHERE ... ;
```

#### 35.4.4. Tipos registro

*nome* RECORD;

As variáveis registro são semelhantes às variáveis tipo-linha, mas não possuem uma estrutura pré-definida. Assumem a estrutura da linha para a qual são atribuídas pelo comando SELECT ou FOR. A subestrutura da variável registro pode mudar toda vez que é usada em uma atribuição. Como consequência, antes de ser utilizada em uma atribuição a variável registro não possui subestrutura, e qualquer tentativa de acessar um de seus campos produz um erro em tempo de execução.

Deve ser observado que RECORD não é um tipo de dado real, mas somente uma indicação. Deve-se ter em mente, também, que declarar uma função do PL/pgSQL como retornando o tipo record não é exatamente o mesmo conceito de variável registro, embora a função possa utilizar uma variável registro para armazenar seu resultado. Nos dois casos a verdadeira estrutura da linha é desconhecida quando a função é escrita, mas na função que retorna o tipo record a estrutura verdadeira é determinada quando o comando que faz a chamada é analisado, enquanto uma variável registro pode mudar a sua estrutura de linha em tempo de execução.

#### 35.4.5. RENAME

RENAME *nome\_antigo* TO *novo\_nome*;

O nome de uma variável, registro ou linha pode ser mudado através da instrução RENAME. A utilidade principal é quando NEW ou OLD devem ser referenciados por outro nome dentro da função de gatilho. Consulte também ALIAS.

Exemplos:

```
RENAME id TO id_usuario;
RENAME esta_variavel TO aquela_variavel;
```

**Nota:** RENAME parece estar com problemas desde o PostgreSQL 7.3. A correção possui baixa prioridade, porque o ALIAS cobre a maior parte dos usos práticos do RENAME.

### 35.5. Expressões

Todas as expressões utilizadas nas instruções do PL/pgSQL são processadas utilizando o executor de SQL regular do servidor. Na verdade, um comando como

```
SELECT expressão
```

é executado utilizando o gerenciador da Interface de Programação do Servidor (SPI). Antes da avaliação, as ocorrências de identificadores variáveis do PL/pgSQL são substituídas por parâmetros, e os valores verdadeiros das variáveis são passados para o executor na matriz de parâmetros. Isto permite que o plano de comando para o `SELECT` seja preparado uma única vez, e depois reutilizado nas avaliações seguintes.

A avaliação feita pelo analisador principal do PostgreSQL produz alguns efeitos colaterais na interpretação de valores constantes. Visto em detalhes existe diferença entre o que estas duas funções fazem:

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS timestamp AS $$
BEGIN
    INSERT INTO logtable VALUES (logtxt, 'now');
    RETURN 'now';
END;
$$ LANGUAGE plpgsql;
```

e

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS timestamp AS $$
DECLARE
    curtime timestamp;
BEGIN
    curtime := 'now';
    INSERT INTO logtable VALUES (logtxt, curtime);
    RETURN curtime;
END;
$$ LANGUAGE plpgsql;
```

No caso da função `logfunc1`, o analisador principal do PostgreSQL sabe, ao preparar o plano para o comando `INSERT`, que a cadeia de caracteres `'now'` deve ser interpretada como `timestamp`, porque a coluna de destino na tabela `logtable` é deste tipo, e por isso cria uma constante a partir de `'now'` contendo a data e hora da análise. Depois, esta constante é utilizada em todas as chamadas à função `logfunc1` durante toda a sessão. É desnecessário dizer que não é este o comportamento o desejado pelo programador.

No caso da função `logfunc2`, o analisador principal do PostgreSQL não sabe o tipo que `'now'` deve se tornar e, portanto, retorna um valor de dado do tipo `text` contendo a cadeia de caracteres `now`. Durante as atribuições seguintes à variável local `curtime`, o interpretador do PL/pgSQL irá converter a cadeia de caracteres para o tipo `timestamp`, chamando as funções `text_out` e `timestamp_in` para fazer a conversão. Portanto, a data e hora computada é atualizada a cada execução, como esperado pelo programador.

A natureza mutável das variáveis registro também apresenta um problema semelhante. Quando campos de uma variável registro são utilizados em expressões ou instruções, os tipos de dado dos campos não devem mudar entre chamadas à mesma expressão, uma vez que a expressão é planejada utilizando o tipo de dado presente quando a expressão é encontrada pela primeira vez. Deve-se ter isto em mente ao escrever procedimentos de gatilhos que tratam eventos para mais de uma tabela; quando for necessário, pode ser utilizado `EXECUTE` para evitar este problema.

## 35.6. Instruções básicas

Esta seção e as seguintes descrevem todos os tipos de instruções compreendidas explicitamente pelo PL/pgSQL. Tudo que não é reconhecido como um destes tipos de instrução é assumido como sendo um comando SQL, e enviado para ser executado pela máquina de banco de dados principal (após a substituição das variáveis do PL/pgSQL na instrução). Desta maneira, por exemplo, os comandos SQL `INSERT`, `UPDATE` e `DELETE` podem ser considerados como sendo instruções da linguagem PL/pgSQL, mas não são listados aqui.

### 35.6.1. Atribuições

A atribuição de um valor a uma variável, ou a um campo de linha ou de registro, é escrita da seguinte maneira:

```
identificador := expressão;
```

Conforme explicado anteriormente, a expressão nesta instrução é avaliada através de um comando `SELECT` do SQL enviado para a máquina de banco de dados principal. A expressão deve produzir um único valor.

Se o tipo de dado do resultado da expressão não corresponder ao tipo de dado da variável, ou se a variável possuir um tipo/precisão específico (como `char(20)`), o valor do resultado será convertido implicitamente pelo interpretador do PL/pgSQL, utilizando a função de saída do tipo do resultado e a função de entrada do tipo da variável. Deve ser observado que este procedimento pode ocasionar erros em tempo de execução gerados pela função de entrada, se a forma cadeia de caracteres do valor do resultado não puder ser aceita pela função de entrada.

Exemplos:

```
id_usuario := 20;
taxa := subtotal * 0.06;
```

### 35.6.2. SELECT INTO

O resultado de um comando `SELECT` que retorna várias colunas (mas apenas uma linha) pode ser atribuído a uma variável registro, a uma variável tipo-linha, ou a uma lista de variáveis escalares. É feito através de

```
SELECT INTO destino expressões_de_seleção FROM ...;
```

onde *destino* pode ser uma variável registro, uma variável linha, ou uma lista separada por vírgulas de variáveis simples e campos de registro/linha. A *expressões\_de\_seleção* e o restante do comando são os mesmos que no SQL comum.

Deve ser observado que é bem diferente da interpretação normal de `SELECT INTO` feita pelo PostgreSQL, onde o destino de `INTO` é uma nova tabela criada. Se for desejado criar uma tabela dentro de uma função PL/pgSQL a partir do resultado do `SELECT`, deve ser utilizada a sintaxe `CREATE TABLE ... AS SELECT`.

Se for utilizado como destino uma linha ou uma lista de variáveis, os valores selecionados devem corresponder exatamente à estrutura do destino, senão ocorre um erro em tempo de execução. Quando o destino é uma variável registro, esta se autoconfigura automaticamente para o tipo linha das colunas do resultado da consulta.

Exceto pela cláusula `INTO`, a instrução `SELECT` é idêntica ao comando `SELECT` normal do SQL, podendo utilizar todos os seus recursos.

A cláusula `INTO` pode aparecer em praticamente todos os lugares na instrução `SELECT`. Habitualmente é escrita logo após o `SELECT`, conforme mostrado acima, ou logo antes do `FROM` — ou seja, logo antes ou logo após a lista de *expressões\_de\_seleção*.

Se a consulta não retornar nenhuma linha, são atribuídos valores nulos aos destinos. Se a consulta retornar várias linhas, a primeira linha é atribuída aos destinos e as demais são desprezadas; deve ser observado que “a primeira linha” não é bem definida a não ser que seja utilizado `ORDER BY`.

A variável especial `FOUND` pode ser verificada imediatamente após a instrução `SELECT INTO` para determinar se a atribuição foi bem-sucedida, ou seja, foi retornada pelo menos uma linha pela consulta. (consulte a Seção 35.6.6). Por exemplo:

```
SELECT INTO meu_registro * FROM emp WHERE nome_emp = meu_nome;
IF NOT FOUND THEN
    RAISE EXCEPTION 'não foi encontrado o empregado %!', meu_nome;
END IF;
```

Para testar se o resultado do registro/linha é nulo, pode ser utilizada a condição `IS NULL`. Entretanto, não existe maneira de saber se foram desprezadas linhas adicionais. A seguir está mostrado um exemplo que trata o caso onde não foi retornada nenhuma linha:

```
DECLARE
    registro_usuario RECORD;
BEGIN
    SELECT INTO registro_usuario * FROM usuarios WHERE id_usuario=3;

    IF registro_usuario.pagina_web IS NULL THEN
        -- o usuario não informou a página na web, retornar "http://"
        RETURN 'http://';
```

```
END IF;
END;
```

### 35.6.3. Execução de expressão ou de consulta sem resultado

Algumas vezes se deseja avaliar uma expressão ou comando e desprezar o resultado (normalmente quando está sendo chamada uma função que produz efeitos colaterais, mas não possui nenhum valor de resultado útil). Para se fazer isto no PL/pgSQL é utilizada a instrução `PERFORM`:

```
PERFORM comando;
```

Esta instrução executa o *comando* e despreza o resultado. A instrução deve ser escrita da mesma maneira que se escreve um comando `SELECT` do SQL, mas com a palavra chave inicial `SELECT` substituída por `PERFORM`. As variáveis da linguagem PL/pgSQL são substituídas no comando da maneira usual. Além disso, a variável especial `FOUND` é definida como verdade se a instrução produzir pelo menos uma linha, ou falso se não produzir nenhuma linha.

**Nota:** Poderia se esperar que `SELECT` sem a cláusula `INTO` produzisse o mesmo resultado, mas atualmente a única forma aceita para isto ser feito é através do `PERFORM`.

Exemplo:

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

### 35.6.4. Não fazer nada

Algumas vezes uma instrução que não faz nada é útil. Por exemplo, pode indicar que uma ramificação da cadeia `if/then/else` é deliberadamente vazia. Para esta finalidade deve ser utilizada a instrução `NULL`:

```
NULL;
```

Por exemplo, os dois fragmentos de código a seguir são equivalentes:

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        NULL; -- ignorar o erro
END;

BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN -- ignorar o erro
END;
```

Qual dos dois escolher é uma questão de gosto.

**Nota:** Na linguagem PL/SQL do Oracle não é permitida instrução vazia e, portanto, a instrução `NULL` é *requerida* em situações como esta. Mas a linguagem PL/pgSQL permite que simplesmente não se escreva nada.

### 35.6.5. Execução de comandos dinâmicos

As vezes é necessário gerar comandos dinâmicos dentro da função PL/pgSQL, ou seja, comandos que envolvem tabelas diferentes ou tipos de dado diferentes cada vez que são executados. A tentativa normal do PL/pgSQL de colocar planos para os comandos no `cache` não funciona neste cenário. A instrução `EXECUTE` é fornecida para tratar este tipo de problema:

```
EXECUTE cadeia_de_caracteres_do_comando;
```

onde *cadeia\_de\_caracteres\_do\_comando* é uma expressão que produz uma cadeia de caracteres (do tipo `text`) contendo o comando a ser executado. A cadeia de caracteres é enviada literalmente para a máquina SQL.



Em particular, deve-se observar que não é feita a substituição das variáveis do PL/pgSQL na cadeia de caracteres do comando. Os valores das variáveis devem ser inseridos na cadeia de caracteres do comando quando esta é construída.

Diferentemente de todos os outros comandos do PL/pgSQL, o comando executado pela instrução `EXECUTE` não é preparado e salvo apenas uma vez por todo o tempo de duração da sessão. Em vez disso, o comando é preparado cada vez que a instrução é executada. A cadeia de caracteres do comando pode ser criada dinamicamente dentro da função para realizar ações em tabelas e colunas diferentes.

Os resultados dos comandos `SELECT` são desprezados pelo `EXECUTE` e, atualmente, o `SELECT INTO` não é suportado pelo `EXECUTE`. Portanto não há maneira de extrair o resultado de um comando `SELECT` criado dinamicamente utilizando o comando `EXECUTE` puro. Entretanto, há duas outras maneiras disto ser feito: uma é utilizando o laço `FOR-IN-EXECUTE` descrito na Seção 35.7.4, e a outra é utilizando um cursor com `OPEN-FOR-EXECUTE`, conforme descrito na Seção 35.8.2.

Quando se trabalha com comandos dinâmicos, muitas vezes é necessário tratar o escape dos apóstrofes. O método recomendado para delimitar texto fixo no corpo da função é utilizar o cifrão (Caso exista código legado que não utiliza a delimitação por cifrão por favor consulte a visão geral na Seção 35.2.1, que pode ajudar a reduzir o esforço para converter este código em um esquema mais razoável).

Os valores dinâmicos a serem inseridos nos comandos construídos requerem um tratamento especial, uma vez que estes também podem conter apóstrofes ou aspas. Um exemplo (assumindo que está sendo utilizada a delimitação por cifrão para a função como um todo e, portanto, os apóstrofes não precisam ser duplicados) é:

```
EXECUTE 'UPDATE tbl SET '
      || quote_ident(nome_da_coluna)
      || ' = '
      || quote_literal(novo_valor)
      || ' WHERE key = '
      || quote_literal(valor_chave);
```

Este exemplo mostra o uso das funções `quote_ident(text)` e `quote_literal(text)`. Por motivo de segurança, as variáveis contendo identificadores de coluna e de tabela devem ser passadas para a função `quote_ident`. As variáveis contendo valores que devem se tornar literais cadeia de caracteres no comando construído devem ser passadas para função `quote_literal`. Estas duas funções executam os passos apropriados para retornar o texto de entrada envolto por aspas ou apóstrofes, respectivamente, com todos os caracteres especiais presentes devidamente colocados em seqüências de escape.

Deve ser observado que a delimitação por cifrão somente é útil para delimitar texto fixo. Seria uma péssima idéia tentar codificar o exemplo acima na forma

```
EXECUTE 'UPDATE tbl SET '
      || quote_ident(nome_da_coluna)
      || ' = $$'
      || novo_valor
      || '$$ WHERE key = '
      || quote_literal(valor_chave);
```

porque não funcionaria se o conteúdo de `novo_valor` tivesse `$$`. A mesma objeção se aplica a qualquer outra delimitação por cifrão escolhida. Portanto, para delimitar texto que não é previamente conhecido *deve* ser utilizada a função `quote_literal`.

Pode ser visto no Exemplo 35-8, onde é construído e executado um comando `CREATE FUNCTION` para definir uma nova função, um caso muito maior de comando dinâmico e `EXECUTE`.

### 35.6.6. Obtenção do status do resultado

Existem diversas maneiras de determinar o efeito de um comando. O primeiro método é utilizar o comando `GET DIAGNOSTICS`, que possui a forma:

```
GET DIAGNOSTICS variável = item [ , ... ] ;
```

Este comando permite obter os indicadores de status do sistema. Cada *item* é uma palavra chave que identifica o valor de estado a ser atribuído a variável especificada (que deve ser do tipo de dado correto para poder receber o valor). Os itens de status disponíveis atualmente são `ROW_COUNT`, o número de linhas processadas pelo último comando SQL enviado para a

máquina SQL, e `RESULT_OID`, o OID da última linha inserida pelo comando SQL mais recente. Deve ser observado que `RESULT_OID` só tem utilidade após um comando `INSERT`.

Exemplo:

```
GET DIAGNOSTICS variavel_inteira = ROW_COUNT;
```

O segundo método para determinar os efeitos de um comando é verificar a variável especial `FOUND`, que é do tipo `boolean`. A variável `FOUND` é iniciada como falso dentro de cada chamada de função PL/pgSQL. É definida por cada um dos seguintes tipos de instrução:

- A instrução `SELECT INTO` define `FOUND` como verdade quando retorna uma linha, e como falso quando não retorna nenhuma linha.
- A instrução `PERFORM` define `FOUND` como verdade quando produz (e despreza) uma linha, e como falso quando não produz nenhuma linha.
- As instruções `UPDATE`, `INSERT` e `DELETE` definem `FOUND` como verdade quando pelo menos uma linha é afetada, e como falso quando nenhuma linha é afetada.
- A instrução `FETCH` define `FOUND` como verdade quando retorna uma linha, e como falso quando não retorna nenhuma linha.
- A instrução `FOR` define `FOUND` como verdade quando interage uma ou mais vezes, senão define como falso. Isto se aplica a todas três variantes da instrução `FOR` (laços `FOR` inteiros, laços `FOR` em conjuntos de registros, e laços `FOR` em conjuntos de registros dinâmicos). A variável `FOUND` é definida desta maneira ao sair do laço `FOR`: dentro da execução do laço a variável `FOUND` não é modificada pela instrução `FOR`, embora possa ser modificada pela execução de outras instruções dentro do corpo do laço.

`FOUND` é uma variável local dentro de cada função PL/pgSQL; qualquer mudança feita na mesma afeta somente a função corrente.

## 35.7. Estruturas de controle

As estruturas de controle provavelmente são a parte mais útil (e mais importante) da linguagem PL/pgSQL. Com as estruturas de controle do PL/pgSQL os dados do PostgreSQL podem ser manipulados de uma forma muito flexível e poderosa.

### 35.7.1. Retorno de uma função

Estão disponíveis dois comandos que permitem retornar dados de uma função: `RETURN` e `RETURN NEXT`.

#### 35.7.1.1. RETURN

```
RETURN expressão;
```

O comando `RETURN` com uma expressão termina a função e retorna o valor da *expressão* para quem chama. Esta forma é utilizada pelas funções do PL/pgSQL que não retornam conjunto.

Qualquer expressão pode ser utilizada para retornar um tipo escalar. O resultado da expressão é automaticamente convertido no tipo de retorno da função conforme descrito nas atribuições. Para retornar um valor composto (linha), deve ser escrita uma variável registro ou linha como a *expressão*.

O valor retornado pela função não pode ser deixado indefinido. Se o controle atingir o final do bloco de nível mais alto da função sem atingir uma instrução `RETURN`, ocorrerá um erro em tempo de execução.

Se a função for declarada como retornando `void`, ainda assim deve ser especificada uma instrução `RETURN`; mas neste caso a expressão após o comando `RETURN` é opcional, sendo ignorada caso esteja presente.

#### 35.7.1.2. RETURN NEXT

```
RETURN NEXT expressão;
```

Quando uma função PL/pgSQL é declarada como retornando `SETOF algum_tipo`, o procedimento a ser seguido é um pouco diferente. Neste caso, os itens individuais a serem retornados são especificados em comandos `RETURN NEXT`, e um

comando `RETURN` final, sem nenhum argumento, é utilizado para indicar que a função chegou ao fim de sua execução. O comando `RETURN NEXT` pode ser utilizado tanto com tipos de dado escalares quanto compostos; no último caso toda uma “tabela” de resultados é retornada.

As funções que utilizam `RETURN NEXT` devem ser chamadas da seguinte maneira:

```
SELECT * FROM alguma_função();
```

Ou seja, a função deve ser utilizada como uma fonte de tabela na cláusula `FROM`.

Na verdade, o comando `RETURN NEXT` não faz o controle sair da função: simplesmente salva o valor da expressão. Em seguida, a execução continua na próxima instrução da função PL/pgSQL. O conjunto de resultados é construído se executando comandos `RETURN NEXT` sucessivos. O `RETURN` final, que não deve possuir argumentos, faz o controle sair da função.

**Nota:** A implementação atual de `RETURN NEXT` para o PL/pgSQL armazena todo o conjunto de resultados antes de retornar da função, conforme foi mostrado acima. Isto significa que, se a função PL/pgSQL produzir um conjunto de resultados muito grande, o desempenho será ruim: os dados serão escritos em disco para evitar exaurir a memória, mas a função não retornará antes que todo o conjunto de resultados tenha sido gerado. Uma versão futura do PL/pgSQL deverá permitir aos usuários definirem funções que retornam conjuntos que não tenham esta limitação. Atualmente, o ponto onde os dados começam a ser escritos em disco é controlado pela variável de configuração `work_mem`. Os administradores que possuem memória suficiente para armazenar conjuntos de resultados maiores, devem considerar o aumento deste parâmetro.

### 35.7.2. Condicionais

As instruções `IF` permitem executar os comandos com base em certas condições. A linguagem PL/pgSQL possui cinco formas de `IF`:

- `IF ... THEN`
- `IF ... THEN ... ELSE`
- `IF ... THEN ... ELSE IF`
- `IF ... THEN ... ELSIF ... THEN ... ELSE`
- `IF ... THEN ... ELSEIF ... THEN ... ELSE`

#### 35.7.2.1. IF-THEN

```
IF expressão_booleana THEN
    instruções
END IF;
```

As instruções `IF-THEN` são a forma mais simples de `IF`. As instruções entre o `THEN` e o `END IF` são executadas se a condição for verdade. Senão, são saltadas.

Exemplo:

```
IF v_id_usuario <> 0 THEN
    UPDATE usuarios SET email = v_email WHERE id_usuario = v_id_usuario;
END IF;
```

#### 35.7.2.2. IF-THEN-ELSE

```
IF expressão_booleana THEN
    instruções
ELSE
    instruções
END IF;
```

As instruções `IF-THEN-ELSE` ampliam o `IF-THEN` permitindo especificar um conjunto alternativo de instruções a serem executadas se a condição for avaliada como falsa.

Exemplos:

```

IF id_pais IS NULL OR id_pais = ''
THEN
    RETURN nome_completo;
ELSE
    RETURN hp_true_filename(id_pais) || '/' || nome_completo;
END IF;

IF v_contador > 0 THEN
    INSERT INTO contador_de_usuários (contador) VALUES (v_contador);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;

```

### 35.7.2.3. IF-THEN-ELSE IF

As instruções IF podem ser aninhadas, como no seguinte exemplo:

```

IF linha_demo.sexo = 'm' THEN
    sexo_extenso := 'masculino';
ELSE
    IF linha_demo.sexo = 'f' THEN
        sexo_extenso := 'feminino';
    END IF;
END IF;

```

Na verdade, quando esta forma é utilizada uma instrução IF está sendo aninhada dentro da parte ELSE da instrução IF externa. Portanto, há necessidade de uma instrução END IF para cada IF aninhado, mais um para o IF-ELSE pai. Embora funcione, cresce de forma tediosa quando existem muitas alternativas a serem verificadas. Por isso existe a próxima forma.

### 35.7.2.4. IF-THEN-ELSIF-ELSE

```

IF expressão_booleana THEN
    instruções
[ ELSEIF expressão_booleana THEN
    instruções
[ ELSEIF expressão_booleana THEN
    instruções
...]]
[ ELSE
    instruções ]
END IF;

```

A instrução IF-THEN-ELSIF-ELSE fornece um método mais conveniente para verificar muitas alternativas em uma instrução. Formalmente equivale aos comandos IF-THEN-ELSE-IF-THEN aninhados, mas somente necessita de um END IF.

Abaixo segue um exemplo:

```

IF numero = 0 THEN
    resultado := 'zero';
ELSIF numero > 0 THEN
    resultado := 'positivo';
ELSIF numero < 0 THEN
    resultado := 'negativo';
ELSE
    -- hmm, a única outra possibilidade é que o número seja nulo
    resultado := 'NULL';
END IF;

```

### 35.7.2.5. IF-THEN-ELSEIF-ELSE

ELSEIF é um aliás para ELSIF.

### 35.7.3. Laços simples

Com as instruções LOOP, EXIT, WHILE e FOR pode-se fazer uma função PL/pgSQL repetir uma série de comandos.

#### 35.7.3.1. LOOP

```
[<<rótulo>>]
LOOP
    instruções
END LOOP;
```

A instrução LOOP define um laço incondicional, repetido indefinidamente até ser terminado por uma instrução EXIT ou RETURN. Nos laços aninhados pode ser utilizado um rótulo opcional na instrução EXIT para especificar o nível de aninhamento que deve ser terminado.

#### 35.7.3.2. EXIT

```
EXIT [ rótulo ] [ WHEN expressão ];
```

Se não for especificado nenhum *rótulo*, o laço mais interno é terminado, e a instrução após o END LOOP é executada a seguir. Se o *rótulo* for especificado, este deve ser o rótulo do nível corrente ou de algum nível externo do laço ou bloco aninhado. Neste caso o laço ou bloco é terminado, e o controle continua na instrução após o END do laço ou do bloco.

Quando WHEN está presente, a saída do laço ocorre somente se a condição especificada for verdadeira, senão o controle passa para a instrução após o EXIT.

Pode ser utilizado EXIT para causar uma saída prematura de qualquer tipo de laço; não está limitado aos laços incondicionais.

Exemplos:

```
LOOP
    -- algum processamento
    IF contador > 0 THEN
        EXIT; -- sair do laço
    END IF;
END LOOP;

LOOP
    -- algum processamento
    EXIT WHEN contador > 0; -- mesmo resultado do exemplo acima
END LOOP;

BEGIN
    -- algum processamento
    IF estoque > 100000 THEN
        EXIT; -- causa a saída do bloco BEGIN
    END IF;
END;
```

#### 35.7.3.3. WHILE

```
[<<rótulo>>]
WHILE expressão LOOP
    instruções
END LOOP;
```

A instrução WHILE repete uma sequência de instruções enquanto a expressão de condição for avaliada como verdade. A condição é verificada logo antes de cada entrada no corpo do laço.

Por exemplo:

```
WHILE quantia_devida > 0 AND saldo_do_certificado_de_bonus > 0 LOOP
    -- algum processamento
END LOOP;
```

```
WHILE NOT expressão_booleana LOOP
    -- algum processamento
END LOOP;
```

#### 35.7.3.4. FOR (variação inteira)

```
[<<rótulo>>]
FOR nome IN [ REVERSE ] expressão .. expressão LOOP
    instruções
END LOOP;
```

Esta forma do FOR cria um laço que interage num intervalo de valores inteiros. A variável *nome* é definida automaticamente como sendo do tipo `integer`, e somente existe dentro do laço. As duas expressões que fornecem o limite inferior e superior do intervalo são avaliadas somente uma vez, ao entrar no laço. Normalmente o passo da interação é 1, mas quando `REVERSE` é especificado se torna -1.

Alguns exemplos de laços FOR inteiros:

```
FOR i IN 1..10 LOOP
    -- algum processamento
    RAISE NOTICE 'i é %', i;
END LOOP;
```

```
FOR i IN REVERSE 10..1 LOOP
    -- algum processamento
END LOOP;
```

Se o limite inferior for maior do que o limite superior (ou menor, no caso do `REVERSE`), o corpo do laço não é executado nenhuma vez. Nenhum erro é gerado.

#### 35.7.4. Laço através do resultado da consulta

Utilizando um tipo diferente de laço FOR, é possível interagir através do resultado de uma consulta e manipular os dados. A sintaxe é:

```
[<<rótulo>>]
FOR registro_ou_linha IN comando LOOP
    instruções
END LOOP;
```

Cada linha de resultado do *comando* (que deve ser um `SELECT`) é atribuída, sucessivamente, à variável registro ou linha, e o corpo do laço é executado uma vez para cada linha. Abaixo segue um exemplo:

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    PERFORM cs_log('Atualização das visões materializadas...');

    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP

        -- Agora "mviews" possui um registro de cs_materialized_views

        PERFORM cs_log('Atualizando a visão materializada ' || quote_ident(mviews.mv_name)
                                                                || ' ...');
        EXECUTE 'TRUNCATE TABLE ' || quote_ident(mviews.mv_name);
        EXECUTE 'INSERT INTO ' || quote_ident(mviews.mv_name) || ' ' || mviews.mv_query;
```

```

END LOOP;

PERFORM cs_log('Fim da atualização das visões materializadas.');
```

RETURN 1;

```

END;
$$ LANGUAGE plpgsql;
```

Se o laço for terminado por uma instrução EXIT, o último valor de linha atribuído ainda é acessível após o laço.

A instrução FOR-IN-EXECUTE é outra forma de interagir sobre linhas:

```

[<rótulo>]
FOR registro_ou_linha IN EXECUTE texto_da_expressão LOOP
    instruções
END LOOP;
```

Esta forma é semelhante à anterior, exceto que o código fonte da instrução SELECT é especificado como uma expressão cadeia de caracteres, que é avaliada e replanejada a cada entrada no laço FOR. Isto permite ao programador escolher entre a velocidade da consulta pré-planejada e a flexibilidade da consulta dinâmica, da mesma maneira que na instrução EXECUTE pura.

**Nota:** Atualmente o analisador da linguagem PL/pgSQL faz distinção entre os dois tipos de laços FOR (inteiro e resultado de consulta), verificando se aparece .. fora de parênteses entre IN e LOOP. Se não for encontrado .., então o laço é assumido como sendo um laço sobre linhas. Se .. for escrito de forma errada, pode causar uma reclamação informando que “a variável do laço, para laço sobre linhas, deve ser uma variável registro ou linha”, em vez de um simples erro de sintaxe como poderia se esperar.

### 35.7.5. Captura de erros

Por padrão, qualquer erro que ocorra em uma função PL/pgSQL interrompe a execução da função, e também da transação envoltória. É possível capturar e se recuperar de erros utilizando um bloco BEGIN com a cláusula EXCEPTION. A sintaxe é uma extensão da sintaxe normal do bloco BEGIN:

```

[ <rótulo> ]
[ DECLARE
    declarações ]
BEGIN
    instruções
EXCEPTION
    WHEN condição [ OR condição ... ] THEN
        instruções_do_tratador
    [ WHEN condição [ OR condição ... ] THEN
        instruções_do_tratador
        ... ]
END;
```

Caso não ocorra nenhum erro, esta forma do bloco simplesmente executa todas as *instruções*, e depois o controle passa para a instrução seguinte ao END. Mas se acontecer algum erro dentro de *instruções*, o processamento das *instruções* é abandonado e o controle passa para a lista de EXCEPTION. É feita a procura na lista da primeira *condição* correspondendo ao erro encontrado. Se for encontrada uma correspondência, as *instruções\_do\_tratador* correspondentes são executadas, e o controle passa para a instrução seguinte ao END. Se não for encontrada nenhuma correspondência, o erro se propaga para fora como se a cláusula EXCEPTION não existisse: o erro pode ser capturado por um bloco envoltório contendo EXCEPTION e, se não houver nenhum, o processamento da função é interrompido.

O nome da *condição* pode ser qualquer um dos mostrados no Apêndice A. Um nome de categoria corresponde a qualquer erro desta categoria. O nome de condição especial OTHERS corresponde a qualquer erro, exceto QUERY\_CANCELED (É possível, mas geralmente não aconselhável, capturar QUERY\_CANCELED por nome). Não há diferença entre letras maiúsculas e minúsculas nos nomes das condições.

Caso ocorra um novo erro dentro das *instruções\_do\_tratador* selecionadas, este não poderá ser capturado por esta cláusula EXCEPTION, mas é propagado para fora. Uma cláusula EXCEPTION envoltória pode capturá-lo.

Quando um erro é capturado pela cláusula `EXCEPTION`, as variáveis locais da função PL/pgSQL permanecem como estavam quando o erro ocorreu, mas todas as modificações no estado persistente do banco de dados dentro do bloco são desfeitas. Como exemplo, consideremos este fragmento de código:

```
INSERT INTO minha_tabela(nome, sobrenome) VALUES('Tom', 'Jones');
BEGIN
    UPDATE minha_tabela SET nome = 'Joe' WHERE sobrenome = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'capturado division_by_zero';
        RETURN x;
END;
```

Quando o controle chegar à atribuição de `y`, vai falhar com um erro de `division_by_zero`. Este erro será capturado pela cláusula `EXCEPTION`. O valor retornado na instrução `RETURN` será o valor de `x` incrementado, mas os efeitos do comando `UPDATE` foram desfeitos. Entretanto, o comando `INSERT` que precede o bloco não é desfeito e, portanto, o resultado final no banco de dados é Tom Jones e não Joe Jones.

**Dica:** Custa significativamente mais entrar e sair de um bloco que contém a cláusula `EXCEPTION` que de um bloco que não contém esta cláusula. Portanto, a cláusula `EXCEPTION` só deve ser utilizada quando for necessária.

## 35.8. Cursores

Em vez de executar toda a consulta de uma vez, é possível definir um *cursor* encapsulando a consulta e, depois, ler umas poucas linhas do resultado da consulta de cada vez. Um dos motivos de se fazer desta maneira, é para evitar o uso excessivo de memória quando o resultado contém muitas linhas (Entretanto, normalmente não há necessidade dos usuários da linguagem PL/pgSQL se preocuparem com isto, uma vez que os laços `FOR` utilizam internamente um cursor para evitar problemas de memória, automaticamente). Uma utilização mais interessante é retornar a referência a um cursor criado pela função, permitindo a quem chamou ler as linhas. Esta forma proporciona uma maneira eficiente para a função retornar conjuntos grandes de linhas.

### 35.8.1. Declaração de variável cursor

Todos os acessos aos cursores na linguagem PL/pgSQL são feitos através de variáveis cursor, que sempre são do tipo de dado especial `refcursor`. Uma forma de criar uma variável cursor é simplesmente declará-la como sendo do tipo `refcursor`. Outra forma é utilizar a sintaxe de declaração de cursor, cuja forma geral é:

```
nome CURSOR [ ( argumentos ) ] FOR comando ;
```

(O `FOR` pode ser substituído por `IS` para ficar compatível com o Oracle). Os *argumentos*, quando especificados, são uma lista separada por vírgulas de pares *nome tipo\_de\_dado*. Esta lista define nomes a serem substituídos por valores de parâmetros na consulta. Os valores verdadeiros que substituirão estes nomes são especificados posteriormente, quando o cursor for aberto.

Alguns exemplos:

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (chave integer) IS SELECT * FROM tenk1 WHERE unicol = chave;
```

Todas estas três variáveis possuem o tipo de dado `refcursor`, mas a primeira pode ser utilizada em qualquer consulta, enquanto a segunda possui uma consulta totalmente especificada *ligada* à mesma, e a terceira possui uma consulta parametrizada ligada à mesma (O parâmetro `chave` será substituído por um valor inteiro quando o cursor for aberto). A variável `curs1` é dita como *desligada* (`unbound`), uma vez que não está ligada a uma determinada consulta.



### 35.8.2. Abertura de cursor

Antes do cursor poder ser utilizado para trazer linhas, este deve ser *aberto* (É a ação equivalente ao comando SQL `DECLARE CURSOR`). A linguagem PL/pgSQL possui três formas para a instrução `OPEN`, duas das quais utilizam variáveis cursor desligadas, enquanto a terceira utiliza uma variável cursor ligada.

#### 35.8.2.1. OPEN FOR SELECT

```
OPEN cursor_desligado FOR SELECT ...;
```

A variável cursor é aberta e recebe a consulta especificada para executar. O cursor não pode estar aberto, e deve ter sido declarado como um cursor desligado, ou seja, simplesmente como uma variável do tipo `refcursor`. O comando `SELECT` é tratado da mesma maneira que nas outras instruções `SELECT` da linguagem PL/pgSQL: Os nomes das variáveis da linguagem PL/pgSQL são substituídos, e o plano de execução é colocado no `cache` para uma possível reutilização.

Exemplo:

```
OPEN curs1 FOR SELECT * FROM foo WHERE chave = minha_chave;
```

#### 35.8.2.2. OPEN FOR EXECUTE

```
OPEN cursor_desligado FOR EXECUTE cadeia_de_caracteres_da_consulta;
```

A variável cursor é aberta e recebe a consulta especificada para executar. O cursor não pode estar aberto, e deve ter sido declarado como um cursor desligado, ou seja, simplesmente como uma variável do tipo `refcursor`. A consulta é especificada como uma expressão cadeia de caracteres da mesma maneira que no comando `EXECUTE`. Como habitual, esta forma provê flexibilidade e, portanto, a consulta pode variar entre execuções.

Exemplo:

```
OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident($1);
```

#### 35.8.2.3. Abertura de cursor ligado

```
OPEN cursor_ligado [ ( valores_dos_argumentos ) ];
```

Esta forma do `OPEN` é utilizada para abrir uma variável cursor cuja consulta foi ligada à mesma ao ser declarada. O cursor não pode estar aberto. Deve estar presente uma lista de expressões com os valores reais dos argumentos se, e somente se, o cursor for declarado como recebendo argumentos. Estes valores são substituídos na consulta. O plano de comando do cursor ligado é sempre considerado como passível de ser colocado no `cache`; neste caso não há forma `EXECUTE` equivalente.

Exemplos:

```
OPEN curs2;
OPEN curs3(42);
```

### 35.8.3. Utilização de cursores

Uma vez que o cursor tenha sido aberto, este pode ser manipulado pelas instruções descritas a seguir.

Para começar, não há necessidade destas manipulações estarem na mesma função que abriu o cursor. Pode ser retornado pela função um valor `refcursor`, e deixar por conta de quem chamou operar o cursor (Internamente, o valor de `refcursor` é simplesmente uma cadeia de caracteres com o nome do tão falado portal que contém a consulta ativa para o cursor. Este nome pode ser passado, atribuído a outras variáveis `refcursor`, e por aí em diante, sem perturbar o portal).

Todos os portais são fechados implicitamente ao término da transação. Portanto, o valor de `refcursor` pode ser utilizado para fazer referência a um cursor aberto até o fim da transação.

#### 35.8.3.1. FETCH

```
FETCH cursor INTO destino;
```

A instrução `FETCH` coloca a próxima linha do cursor no destino, que pode ser uma variável linha, uma variável registro, ou uma lista separada por vírgulas de variáveis simples, da mesma maneira que no `SELECT INTO`. Como no `SELECT INTO`, pode ser verificada a variável especial `FOUND` para ver se foi obtida uma linha, ou não.

Exemplos:

```
FETCH curs1 INTO variável_linha;
FETCH curs2 INTO foo, bar, baz;
```

### 35.8.3.2. CLOSE

```
CLOSE cursor;
```

A instrução `CLOSE` fecha o portal subjacente ao cursor aberto. Pode ser utilizada para liberar recursos antes do fim da transação, ou para liberar a variável cursor para que esta possa ser aberta novamente.

Exemplo:

```
CLOSE curs1;
```

### 35.8.3.3. Retornar cursor

As funções PL/pgSQL podem retornar cursores para quem fez a chamada. É útil para retornar várias linhas ou colunas, especialmente em conjuntos de resultados muito grandes. Para ser feito, a função abre o cursor e retorna o nome do cursor para quem chamou (ou simplesmente abre o cursor utilizando o nome do portal especificado por, ou de outra forma conhecido por, quem chamou). Quem chamou poderá então ler as linhas usando o cursor. O cursor pode ser fechado por quem chamou, ou será fechado automaticamente ao término da transação.

O nome do portal utilizado para o cursor pode ser especificado pelo programador ou gerado automaticamente. Para especificar o nome do portal deve-se, simplesmente, atribuir uma cadeia de caracteres à variável `refcursor` antes de abri-la. O valor cadeia de caracteres da variável `refcursor` será utilizado pelo `OPEN` como o nome do portal subjacente. Entretanto, quando a variável `refcursor` é nula, o `OPEN` gera automaticamente um nome que não conflita com nenhum portal existente, e atribui este nome à variável `refcursor`.

**Nota:** Uma variável cursor ligada é inicializada com o valor cadeia de caracteres que representa o seu nome e, portanto, o nome do portal é o mesmo da variável cursor, a menos que o programador mude este nome fazendo uma atribuição antes de abrir o cursor. Porém, uma variável cursor desligada tem inicialmente o valor nulo por padrão e, portanto, recebe um nome único gerado automaticamente, a menos que este seja mudado.

O exemplo a seguir mostra uma maneira de fornecer o nome do cursor por quem chama:

```
CREATE TABLE teste (col text);
INSERT INTO teste VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM teste;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funcursor');

    reffunc
-----
funcursor
(1 linha)

FETCH ALL IN funcursor;
```

```
col
-----
123
(1 linha)
```

```
COMMIT;
```

O exemplo a seguir usa a geração automática de nome de cursor:

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM teste;
    RETURN ref;
END;
' LANGUAGE plpgsql;
```

```
BEGIN;
SELECT reffunc2();
```

```
      reffunc2
-----
<unnamed portal 1>
(1 linha)
```

```
FETCH ALL IN "<unnamed cursor 1>";
```

```
col
-----
123
(1 linha)
```

```
COMMIT;
```

Os exemplos a seguir mostram uma maneira de retornar vários cursores de uma única função:

```
CREATE FUNCTION minha_funcao(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM tabela_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM tabela_2;
    RETURN NEXT $2;
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

```
-- é necessário estar em uma transação para poder usar cursor
BEGIN;
```

```
SELECT * FROM minha_funcao('a', 'b');
```

```
FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

## 35.9. Erros e mensagens

A instrução `RAISE` é utilizada para gerar mensagens informativas e causar erros.

```
RAISE nível 'formato' [, variável [, ...]];
```

Os níveis possíveis são `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, e `EXCEPTION`. O nível `EXCEPTION` causa um erro (que normalmente interrompe a transação corrente); os outros níveis apenas geram mensagens com diferentes níveis de prioridade. Se as mensagens de uma determinada prioridade são informadas ao cliente, escritas no `log` do servidor, ou as duas coisas, é controlado pelas variáveis de configuração `log_min_messages` e `client_min_messages`. Para obter informações adicionais deve ser consultada a Seção 16.4.

Dentro da cadeia de caracteres de formatação, o caractere `%` é substituído pela representação na forma de cadeia de caracteres do próximo argumento opcional. Deve ser escrito `%%` para produzir um `%` literal. Deve ser observado que atualmente os argumentos opcionais devem ser variáveis simples, e não expressões, e o formato deve ser um literal cadeia de caracteres simples.

Neste exemplo o valor de `v_job_id` substitui o caractere `%` na cadeia de caracteres:

```
RAISE NOTICE 'Chamando cs_create_job(%)', v_job_id;
```

Este exemplo interrompe a transação com a mensagem de erro fornecida:

```
RAISE EXCEPTION 'ID inexistente --> %', id_usuario;
```

Atualmente `RAISE EXCEPTION` sempre gera o mesmo código `SQLSTATE`, `P0001`, não importando a mensagem com a qual seja chamado. É possível capturar esta exceção com `EXCEPTION ... WHEN RAISE_EXCEPTION THEN ...`, mas não há como diferenciar um `RAISE` de outro.

## 35.10. Procedimentos de gatilho

A linguagem PL/pgSQL pode ser utilizada para definir procedimentos de gatilho. O procedimento de gatilho é criado pelo comando `CREATE FUNCTION`, declarando o procedimento como uma função sem argumentos e que retorna o tipo `trigger`. Deve ser observado que a função deve ser declarada sem argumentos, mesmo que espere receber os argumentos especificados no comando `CREATE TRIGGER` — os argumentos do gatilho são passados através de `TG_ARGV`, conforme descrito abaixo.

Quando uma função escrita em PL/pgSQL é chamada como um gatilho, diversas variáveis especiais são criadas automaticamente no bloco de nível mais alto. São estas:

`NEW`

Tipo de dado `RECORD`; variável contendo a nova linha do banco de dados, para as operações de `INSERT/UPDATE` nos gatilhos no nível de linha. O valor desta variável é `NULL` nos gatilhos no nível de instrução.

`OLD`

Tipo de dado `RECORD`; variável contendo a antiga linha do banco de dados, para as operações de `UPDATE/DELETE` nos gatilhos no nível de linha. O valor desta variável é `NULL` nos gatilhos no nível de instrução.

`TG_NAME`

Tipo de dado `name`; variável contendo o nome do gatilho disparado.

`TG_WHEN`

Tipo de dado `text`; uma cadeia de caracteres contendo `BEFORE` ou `AFTER`, dependendo da definição do gatilho.

`TG_LEVEL`

Tipo de dado `text`; uma cadeia de caracteres contendo `ROW` ou `STATEMENT`, dependendo da definição do gatilho.

`TG_OP`

Tipo de dado `text`; uma cadeia de caracteres contendo `INSERT`, `UPDATE`, ou `DELETE`, informando para qual operação o gatilho foi disparado.

`TG_RELID`

Tipo de dado `oid`; o ID de objeto da tabela que causou o disparo do gatilho.

`TG_RELNAME`

Tipo de dado `name`; o nome da tabela que causou o disparo do gatilho.

TG\_NARGS

Tipo de dado `integer`; o número de argumentos fornecidos ao procedimento de gatilho na instrução `CREATE TRIGGER`.

TG\_ARGV[ ]

Tipo de dado matriz de `text`; os argumentos da instrução `CREATE TRIGGER`. O contador do índice começa por 0. Índices inválidos (menor que 0 ou maior ou igual a `tg_nargs`) resultam em um valor nulo.

Uma função de gatilho deve retornar nulo, ou um valor registro/linha possuindo a mesma estrutura da tabela para a qual o gatilho foi disparado.

Os gatilhos no nível de linha disparados `BEFORE` (antes) podem retornar nulo, para sinalizar ao gerenciador do gatilho para pular o restante da operação para esta linha (ou seja, os gatilhos posteriores não serão disparados, e não ocorrerá o `INSERT/UPDATE/DELETE` para esta linha. Se for retornado um valor diferente de nulo, então a operação prossegue com este valor de linha. Retornar um valor de linha diferente do valor original de `NEW` altera a linha que será inserida ou atualizada (mas não tem efeito direto no caso do `DELETE`). Para alterar a linha a ser armazenada, é possível substituir valores individuais diretamente em `NEW` e retornar o `NEW` modificado, ou construir um novo registro/linha completo a ser retornado.

O valor retornado por um gatilho `BEFORE` ou `AFTER` no nível de instrução, ou por um gatilho `AFTER` no nível de linha, é sempre ignorado; pode muito bem ser nulo. Entretanto, qualquer um destes tipos de gatilho pode interromper toda a operação gerando um erro.

O Exemplo 35-1 mostra um exemplo de procedimento de gatilho escrito em PL/pgSQL.

### Exemplo 35-1. Procedimento de gatilho PL/pgSQL

O gatilho deste exemplo garante que quando é inserida ou atualizada uma linha na tabela, fica sempre registrado nesta linha o usuário que efetuou a inserção ou a atualização, e quando isto ocorreu. Além disso, o gatilho verifica se é fornecido o nome do empregado, e se o valor do salário é um número positivo.

```
CREATE TABLE emp (
    nome_emp      text,
    salario       integer,
    ultima_data   timestamp,
    ultimo_usuario text
);

CREATE FUNCTION emp_gatilho() RETURNS trigger AS $emp_gatilho$
BEGIN
    -- Verificar se foi fornecido o nome e o salário do empregado
    IF NEW.nome_emp IS NULL THEN
        RAISE EXCEPTION 'O nome do empregado não pode ser nulo';
    END IF;
    IF NEW.salario IS NULL THEN
        RAISE EXCEPTION '% não pode ter um salário nulo', NEW.nome_emp;
    END IF;

    -- Quem paga para trabalhar?
    IF NEW.salario < 0 THEN
        RAISE EXCEPTION '% não pode ter um salário negativo', NEW.nome_emp;
    END IF;

    -- Registrar quem alterou a folha de pagamento e quando
    NEW.ultima_data := 'now';
    NEW.ultimo_usuario := current_user;
    RETURN NEW;
END;
$emp_gatilho$ LANGUAGE plpgsql;

CREATE TRIGGER emp_gatilho BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_gatilho();

INSERT INTO emp (nome_emp, salario) VALUES ('João',1000);
```

```
INSERT INTO emp (nome_emp, salario) VALUES ('José',1500);
INSERT INTO emp (nome_emp, salario) VALUES ('Maria',2500);

SELECT * FROM emp;
```

nome_emp	salario	ultima_data	ultimo_usuario
João	1000	2005-11-25 07:07:50.59532	folha
José	1500	2005-11-25 07:07:50.691905	folha
Maria	2500	2005-11-25 07:07:50.694995	folha

(3 linhas)

### Exemplo 35-2. Procedimento de gatilho PL/pgSQL para registrar inserção e atualização

O gatilho deste exemplo garante que quando é inserida ou atualizada uma linha na tabela, fica sempre registrado nesta linha o usuário que efetuou a inserção ou a atualização, e quando isto ocorreu. Porém, diferentemente do gatilho anterior, a criação e a atualização da linha são registradas em colunas diferentes. Além disso, o gatilho verifica se é fornecido o nome do empregado, e se o valor do salário é um número positivo.<sup>1</sup>

```
CREATE TABLE emp (
    nome_emp      text,
    salario       integer,
    usu_cria      text,      -- Usuário que criou a linha
    data_cria     timestamp, -- Data da criação da linha
    usu_atu       text,      -- Usuário que fez a atualização
    data_atu      timestamp  -- Data da atualização
);

CREATE FUNCTION emp_gatilho() RETURNS trigger AS $emp_gatilho$
BEGIN
    -- Verificar se foi fornecido o nome do empregado
    IF NEW.nome_emp IS NULL THEN
        RAISE EXCEPTION 'O nome do empregado não pode ser nulo';
    END IF;
    IF NEW.salario IS NULL THEN
        RAISE EXCEPTION '% não pode ter um salário nulo', NEW.nome_emp;
    END IF;

    -- Quem paga para trabalhar?
    IF NEW.salario < 0 THEN
        RAISE EXCEPTION '% não pode ter um salário negativo', NEW.nome_emp;
    END IF;

    -- Registrar quem criou a linha e quando
    IF (TG_OP = 'INSERT') THEN
        NEW.data_cria := current_timestamp;
        NEW.usu_cria  := current_user;
    -- Registrar quem alterou a linha e quando
    ELSIF (TG_OP = 'UPDATE') THEN
        NEW.data_atu := current_timestamp;
        NEW.usu_atu  := current_user;
    END IF;
    RETURN NEW;
END;
$emp_gatilho$ LANGUAGE plpgsql;

CREATE TRIGGER emp_gatilho BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_gatilho();

INSERT INTO emp (nome_emp, salario) VALUES ('João',1000);
INSERT INTO emp (nome_emp, salario) VALUES ('José',1500);
INSERT INTO emp (nome_emp, salario) VALUES ('Maria',250);
UPDATE emp SET salario = 2500 WHERE nome_emp = 'Maria';
```

```
SELECT * FROM emp;
```

nome_emp	salario	usu_cria	data_cria	usu_atu	data_atu
João	1000	folha	2005-11-25 08:11:40.63868		
José	1500	folha	2005-11-25 08:11:40.674356		
Maria	2500	folha	2005-11-25 08:11:40.679592	folha	2005-11-25 08:11:40.682394

(3 linhas)

Uma outra maneira de registrar as modificações na tabela envolve a criação de uma nova tabela contendo uma linha para cada inserção, atualização ou exclusão que ocorra. Esta abordagem pode ser considerada como uma auditoria das mudanças na tabela. O Exemplo 35-3 mostra um procedimento de gatilho de auditoria em PL/pgSQL.

### Exemplo 35-3. Procedimento de gatilho PL/pgSQL para auditoria

Este gatilho garante que todas as inserções, atualizações e exclusões de uma linha na tabela emp é registrada (isto é, auditada) na tabela emp\_audit. O nome de usuário e a hora corrente são gravadas na linha, junto com o tipo de operação que foi realizada.

```
CREATE TABLE emp (
    nome_emp    text NOT NULL,
    salario     integer
);

CREATE TABLE emp_audit(
    operacao    char(1) NOT NULL,
    usuario     text NOT NULL,
    data        timestamp NOT NULL,
    nome_emp    text NOT NULL,
    salario     integer
);

CREATE OR REPLACE FUNCTION processa_emp_audit() RETURNS TRIGGER AS $emp_audit$
BEGIN
    --
    -- Cria uma linha na tabela emp_audit para refletir a operação
    -- realizada na tabela emp. Utiliza a variável especial TG_OP
    -- para descobrir a operação sendo realizada.
    --
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO emp_audit SELECT 'E', user, now(), OLD.*;
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO emp_audit SELECT 'A', user, now(), NEW.*;
        RETURN NEW;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO emp_audit SELECT 'I', user, now(), NEW.*;
        RETURN NEW;
    END IF;
    RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho AFTER
END;
$emp_audit$ language plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit();

INSERT INTO emp (nome_emp, salario) VALUES ('João',1000);
INSERT INTO emp (nome_emp, salario) VALUES ('José',1500);
INSERT INTO emp (nome_emp, salario) VALUES ('Maria',250);
UPDATE emp SET salario = 2500 WHERE nome_emp = 'Maria';
```

```
DELETE FROM emp WHERE nome_emp = 'João';
```

```
SELECT * FROM emp;
```

```
nome_emp | salario
-----+-----
José     |    1500
Maria    |    2500
(2 linhas)
```

```
SELECT * FROM emp_audit;
```

```
operacao | usuario |          data          | nome_emp | salario
-----+-----+-----+-----+-----
I        | folha   | 2005-11-25 09:06:03.008735 | João    |    1000
I        | folha   | 2005-11-25 09:06:03.014245 | José    |    1500
I        | folha   | 2005-11-25 09:06:03.049443 | Maria   |     250
A        | folha   | 2005-11-25 09:06:03.052972 | Maria   |    2500
E        | folha   | 2005-11-25 09:06:03.056774 | João    |    1000
(5 linhas)
```

#### Exemplo 35-4. Procedimento de gatilho PL/pgSQL para auditoria no nível de coluna

Este gatilho registra todas as atualizações realizadas nas colunas `nome_emp` e `salario` da tabela `emp` na tabela `emp_audit` (isto é, as colunas são auditadas). O nome de usuário e a hora corrente são registrados junto com a chave da linha (`id`) e a informação atualizada. Não é permitido atualizar a chave da linha. Este exemplo difere do anterior pela auditoria ser no nível de coluna, e não no nível de linha.<sup>2</sup>

```
CREATE TABLE emp (
    id          serial PRIMARY KEY,
    nome_emp    text NOT NULL,
    salario     integer
);

CREATE TABLE emp_audit(
    usuario     text NOT NULL,
    data        timestamp NOT NULL,
    id          integer NOT NULL,
    coluna      text NOT NULL,
    valor_antigo text NOT NULL,
    valor_novo  text NOT NULL
);

CREATE OR REPLACE FUNCTION processa_emp_audit() RETURNS TRIGGER AS $emp_audit$
BEGIN
    --
    -- Não permitir atualizar a chave primária
    --
    IF (NEW.id <> OLD.id) THEN
        RAISE EXCEPTION 'Não é permitido atualizar o campo ID';
    END IF;
    --
    -- Inserir linhas na tabela emp_audit para refletir as alterações
    -- realizada na tabela emp.
    --
    IF (NEW.nome_emp <> OLD.nome_emp) THEN
        INSERT INTO emp_audit SELECT current_user, current_timestamp,
                                     NEW.id, 'nome_emp', OLD.nome_emp, NEW.nome_emp;
    END IF;
    IF (NEW.salario <> OLD.salario) THEN
        INSERT INTO emp_audit SELECT current_user, current_timestamp,
                                     NEW.id, 'salario', OLD.salario, NEW.salario;
    END IF;
```



```

        RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho AFTER
    END;
$emp_audit$ language plpgsql;

```

```

CREATE TRIGGER emp_audit
AFTER UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit();

INSERT INTO emp (nome_emp, salario) VALUES ('João',1000);
INSERT INTO emp (nome_emp, salario) VALUES ('José',1500);
INSERT INTO emp (nome_emp, salario) VALUES ('Maria',2500);
UPDATE emp SET salario = 2500 WHERE id = 2;
UPDATE emp SET nome_emp = 'Maria Cecília' WHERE id = 3;
UPDATE emp SET id=100 WHERE id=1;
ERRO: Não é permitido atualizar o campo ID

```

```
SELECT * FROM emp;
```

id	nome_emp	salario
1	João	1000
2	José	2500
3	Maria Cecília	2500

(3 linhas)

```
SELECT * FROM emp_audit;
```

usuario	data	id	coluna	valor_antigo	valor_novo
folha	2005-11-25 12:21:08.493268	2	salario	1500	2500
folha	2005-11-25 12:21:08.49822	3	nome_emp	Maria	Maria Cecília

(2 linhas)

Uma das utilizações de gatilho é para manter uma tabela contendo o sumário de outra tabela. O sumário produzido pode ser utilizado no lugar da tabela original em diversas consultas — geralmente com um tempo de execução bem menor. Esta técnica é muito utilizada em Armazém de Dados (Data Warehousing), onde as tabelas dos dados medidos ou observados (chamadas de tabelas fato) podem ser muito grandes. O Exemplo 35-5 mostra um procedimento de gatilho em PL/pgSQL para manter uma tabela de sumário de uma tabela fato em um armazém de dados.

#### Exemplo 35-5. Procedimento de gatilho PL/pgSQL para manter uma tabela sumário

O esquema que está detalhado a seguir é parcialmente baseado no exemplo *Grocery Store* do livro *The Data Warehouse Toolkit* de Ralph Kimball.

```

--
-- Main tables - time dimension and sales fact.
--
CREATE TABLE time_dimension (
    time_key          integer NOT NULL,
    day_of_week       integer NOT NULL,
    day_of_month       integer NOT NULL,
    month             integer NOT NULL,
    quarter           integer NOT NULL,
    year              integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key          integer NOT NULL,
    product_key       integer NOT NULL,
    store_key         integer NOT NULL,
    amount_sold       numeric(12,2) NOT NULL,

```

```

        units_sold                integer NOT NULL,
        amount_cost               numeric(12,2) NOT NULL
    );
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Summary table - sales by time.
--
CREATE TABLE sales_summary_bytime (
    time_key                    integer NOT NULL,
    amount_sold                 numeric(15,2) NOT NULL,
    units_sold                  numeric(12) NOT NULL,
    amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- Function and trigger to amend summarized column(s) on UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER AS
$maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN

        -- Work out the increment/decrement amount(s).
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;

        ELSIF (TG_OP = 'UPDATE') THEN

            -- forbid updates that change the time_key -
            -- (probably not too onerous, as DELETE + INSERT is how most
            -- changes will be made).
            IF ( OLD.time_key != NEW.time_key) THEN
                RAISE EXCEPTION 'Update of time_key : % -> % not allowed', OLD.time_key,
                    NEW.time_key;
            END IF;

            delta_time_key = OLD.time_key;
            delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
            delta_units_sold = NEW.units_sold - OLD.units_sold;
            delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

        ELSIF (TG_OP = 'INSERT') THEN

            delta_time_key = NEW.time_key;
            delta_amount_sold = NEW.amount_sold;
            delta_units_sold = NEW.units_sold;
            delta_amount_cost = NEW.amount_cost;

        END IF;

        -- Update the summary row with the new values.

```

```

UPDATE sales_summary_bytime
SET amount_sold = amount_sold + delta_amount_sold,
    units_sold = units_sold + delta_units_sold,
    amount_cost = amount_cost + delta_amount_cost
WHERE time_key = delta_time_key;

-- There might have been no row with this time_key (e.g new data!).
IF (NOT FOUND) THEN
BEGIN
    INSERT INTO sales_summary_bytime (
        time_key,
        amount_sold,
        units_sold,
        amount_cost)
    VALUES (
        delta_time_key,
        delta_amount_sold,
        delta_units_sold,
        delta_amount_cost
    );

EXCEPTION
--
-- Catch race condition when two transactions are adding data
-- for a new time_key.
--
WHEN UNIQUE_VIOLATION THEN
    UPDATE sales_summary_bytime
    SET amount_sold = amount_sold + delta_amount_sold,
        units_sold = units_sold + delta_units_sold,
        amount_cost = amount_cost + delta_amount_cost
    WHERE time_key = delta_time_key;

END;
END IF;
RETURN NULL;

END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

```

### Exemplo 35-6. Procedimento de gatilho para controlar sobreposição de datas

O gatilho deste exemplo verifica se o compromisso sendo agendado ou modificado se sobrepõe a outro compromisso já agendado. Se houver sobreposição, emite mensagem de erro e não permite a operação.<sup>3</sup>

Abaixo está mostrado o script utilizado para criar a tabela, a função de gatilho e os gatilhos de inserção e atualização.

```

CREATE TABLE agendamentos (
    id          SERIAL PRIMARY KEY,
    nome        TEXT,
    evento      TEXT,
    data_inicio TIMESTAMP,
    data_fim    TIMESTAMP
);

CREATE FUNCTION fun_verifica_agendamentos() RETURNS "trigger" AS
$fun_verifica_agendamentos$
BEGIN
    /* Verificar se a data de início é maior que a data de fim */

```

```

IF NEW.data_inicio > NEW.data_fim THEN
    RAISE EXCEPTION 'A data de início não pode ser maior que a data de fim';
END IF;
/* Verificar se há sobreposição com agendamentos existentes */
IF EXISTS (
    SELECT 1
    FROM agendamentos
    WHERE nome = NEW.nome
    AND ((data_inicio, data_fim) OVERLAPS
        (NEW.data_inicio, NEW.data_fim))
)
THEN
    RAISE EXCEPTION 'impossível agendar - existe outro compromisso';
END IF;
RETURN NEW;
END;
$fun_verifica_agendamentos$ LANGUAGE plpgsql;

COMMENT ON FUNCTION fun_verifica_agendamentos() IS
    'Verifica se o agendamento é possível';

CREATE TRIGGER trg_agendamentos_ins
    BEFORE INSERT ON agendamentos
    FOR EACH ROW
    EXECUTE PROCEDURE fun_verifica_agendamentos();

CREATE TRIGGER trg_agendamentos_upd
    BEFORE UPDATE ON agendamentos
    FOR EACH ROW
    EXECUTE PROCEDURE fun_verifica_agendamentos();

```

Abaixo está mostrado um exemplo de utilização do gatilho. Deve ser observado que os intervalos ('2005-08-23 14:00:00', '2005-08-23 15:00:00') e ('2005-08-23 15:00:00', '2005-08-23 16:00:00') não se sobrepõem, uma vez que o primeiro intervalo termina às quinze horas, enquanto o segundo intervalo inicia às quinze horas, estando, portanto, o segundo intervalo imediatamente após o primeiro.

```

=> INSERT INTO agendamentos VALUES (DEFAULT, 'Joana', 'Congresso', '2005-08-23', '2005-08-24');
=> INSERT INTO agendamentos VALUES (DEFAULT, 'Joana', 'Viagem', '2005-08-24', '2005-08-26');
=> INSERT INTO agendamentos VALUES (DEFAULT, 'Joana', 'Palestra', '2005-08-23', '2005-08-26');
ERRO: impossível agendar - existe outro compromisso
=> INSERT INTO agendamentos VALUES (DEFAULT, 'Maria', 'Cabeleireiro', '2005-08-23
14:00:00', '2005-08-23 15:00:00');
=> INSERT INTO agendamentos VALUES (DEFAULT, 'Maria', 'Manicure', '2005-08-23 15:00:00', '2005-
08-23 16:00:00');
=> INSERT INTO agendamentos VALUES (DEFAULT, 'Maria', 'Médico', '2005-08-23 14:30:00', '2005-08-
23 15:00:00');
ERRO: impossível agendar - existe outro compromisso
=> UPDATE agendamentos SET data_inicio='2005-08-24' WHERE id=2;
ERRO: impossível agendar - existe outro compromisso
=> SELECT * FROM agendamentos;

```

id	nome	evento	data_inicio	data_fim
1	Joana	Congresso	2005-08-23 00:00:00	2005-08-24 00:00:00
2	Joana	Viagem	2005-08-24 00:00:00	2005-08-26 00:00:00
4	Maria	Cabeleireiro	2005-08-23 14:00:00	2005-08-23 15:00:00
5	Maria	Manicure	2005-08-23 15:00:00	2005-08-23 16:00:00

(4 linhas)

## 35.11. Conversão do PL/SQL do Oracle para o PL/pgSQL do PostgreSQL

Esta seção explica as diferenças entre a linguagem PL/pgSQL do PostgreSQL e a linguagem PL/SQL do Oracle, para ajudar aos desenvolvedores na conversão dos aplicativos do Oracle para o PostgreSQL.

A linguagem PL/pgSQL é semelhante à linguagem PL/SQL em muitos aspectos. É uma linguagem estruturada em blocos, imperativa, e todas as variáveis devem ser declaradas. As atribuições, laços e condicionais são semelhantes. As principais diferenças que se deve ter em mente na conversão da linguagem PL/SQL para a linguagem PL/pgSQL, são:

- No PostgreSQL não existe valor padrão para parâmetros.
- No PostgreSQL os nomes das funções podem ser sobrecarregados. Geralmente isto é utilizado para superar o problema da falta de parâmetros padrão.
- Os cursores não são necessários na linguagem PL/pgSQL, basta por a consulta na instrução `FOR` (Consulte o Exemplo 35-8.)
- No PostgreSQL é necessário utilizar a delimitação por cifrão (\$), ou criar seqüências de escape para os apóstrofes presentes no corpo da função. Consulte a Seção 35.2.1.
- Em vez de pacotes, são utilizados esquemas para organizar as funções em grupos.
- Não existem pacotes, não existem variáveis no nível de pacote também. Isto aborrece um pouco. Em seu lugar, o estado por sessão pode ser mantido em tabelas temporárias.
- Não podem ser utilizados nomes de parâmetros idênticos aos das colunas referenciadas na função. O Oracle permite que isto seja feito se o nome do parâmetro for qualificado na forma `nome_da_função.nome_do_parâmetro`.<sup>4</sup>

### 35.11.1. Exemplos de conversão

O Exemplo 35-7 mostra como converter uma função simples de PL/SQL para PL/pgSQL.

#### Exemplo 35-7. Conversão de uma função simples de PL/SQL para PL/pgSQL

Abaixo está uma função escrita na linguagem PL/SQL do Oracle:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_nome    IN varchar,
                                                    v-versao IN varchar)
RETURN varchar IS
BEGIN
    IF v-versao IS NULL THEN
        RETURN v_nome;
    END IF;
    RETURN v_nome || '/' || v-versao;
END;
/
show errors;
```

Vamos examinar esta função para ver as diferenças com relação à linguagem PL/pgSQL:

- O Oracle permite que sejam passados parâmetros `IN`, `OUT` e `INOUT` para as funções. Por exemplo, `INOUT` significa que o parâmetro recebe um valor e retorna outro. O PostgreSQL somente possui parâmetros `IN` e, portanto, não há especificação do tipo do parâmetro.
- A palavra chave `RETURN` no protótipo da função (não no corpo da função), no PostgreSQL se torna `RETURNS`. Além disso, `IS` se torna `AS`, e é necessário adicionar a cláusula `LANGUAGE`, porque a linguagem PL/pgSQL não é a única possível.
- No PostgreSQL o corpo da função é considerado como sendo um literal cadeia de caracteres, portanto é necessário utilizar delimitação por apóstrofes ou cifrão em torno do corpo da função. Isto substitui a barra (/) terminadora da abordagem do Oracle.
- Não existe o comando `/show errors` no PostgreSQL, e não há necessidade uma vez que os erros são relatados automaticamente.

Abaixo está mostrado como esta função deve se parecer após ser convertida para o PostgreSQL:<sup>5</sup>

```

CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_nome  varchar,
                                                  v-versao varchar)
RETURNS varchar AS $$
BEGIN
    IF v-versao IS NULL THEN
        RETURN v_nome;
    END IF;
    RETURN v_nome || '/' || v-versao;
END;
$$ LANGUAGE plpgsql;

```

O Exemplo 35-8 mostra como converter uma função que cria outra função, e como tratar o problema dos apóstrofes.

### Exemplo 35-8. Conversão de uma função que cria outra função de PL/SQL para PL/pgSQL

O procedimento abaixo obtém linhas a partir de uma instrução `SELECT`, e constrói uma função grande com os resultados em instruções `IF`, por motivo de eficiência. Em particular, deve ser observada a diferença entre o cursor e o laço `FOR`.

Esta é a versão Oracle:

```

CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;

    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR,
        v_domain IN VARCHAR, v_url IN VARCHAR) RETURN VARCHAR IS BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
            ' IF v_' || referrer_key.kind
            || ' LIKE ''' || referrer_key.key_string
            || ''' THEN RETURN ''' || referrer_key.referrer_type
            || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;

```

Abaixo está mostrado como esta função ficaria no PostgreSQL:

```

CREATE OR REPLACE FUNCTION cs_update_referrer_type_proc() RETURNS void AS $func$
DECLARE
    referrer_key RECORD; -- declare a generic record to be used in a FOR
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN' ;

    -- Notice how we scan through the results of a query in a FOR loop
    -- using the FOR <record> construct.

    FOR referrer_key IN SELECT * FROM cs_referrer_keys ORDER BY try_order LOOP
        func_body := func_body ||
            ' IF v_' || referrer_key.kind
            || ' LIKE ' || quote_literal(referrer_key.key_string)
            || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
            || '; END IF;' ;
    END LOOP;

```

```

func_body := func_body || ' RETURN NULL; END;';

func_cmd :=
  'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
                                                    v_domain varchar,
                                                    v_url varchar)
    RETURNS varchar AS '
  || quote_literal(func_body)
  || ' LANGUAGE plpgsql;' ;

EXECUTE func_cmd;
RETURN;
END;
$func$ LANGUAGE plpgsql;

```

Deve ser observado como o corpo da função é construído em separado e passado através da função `quote_literal` para duplicar qualquer apóstrofo porventura existente. Esta técnica é necessária, porque não pode ser utilizada a delimitação por cifrão com segurança para definir a nova função: não há certeza de quais cadeias de caracteres serão interpoladas a partir do campo `referrer_key.key_string` (Aqui é assumido que se pode confiar que `referrer_key.kind` será sempre `host`, `domain`, ou `url`, mas `referrer_key.key_string` pode ser qualquer coisa e, em particular, pode conter o caractere cifrão). Na verdade esta função é uma melhoria com relação à original do Oracle, porque não irá gerar código com erro quando `referrer_key.key_string` ou `referrer_key.referrer_type` contiverem apóstrofes.

O Exemplo 35-9 mostra como converter uma função com parâmetros OUT e manipulação de cadeia de caracteres. O PostgreSQL não possui a função `instr`, mas isto pode ser superado utilizando uma combinação de outras funções. Na Seção 35.11.3 existe uma implementação em PL/pgSQL da função `instr` que pode ser utilizada para facilitar a conversão.

### Exemplo 35-9. Conversão de um procedimento com manipulação de cadeia de caracteres e parâmetros OUT de PL/SQL para PL/pgSQL

O procedimento mostrado abaixo, escrito na linguagem PL/SQL do Oracle, é utilizado para analisar uma URL e retornar vários elementos (hospedeiro, caminho e comando). No PostgreSQL, as funções podem retornar apenas um valor. Uma forma de superar este problema é tornar o valor retornado do tipo composto (tipo linha).

Esta é a versão Oracle:

```

CREATE OR REPLACE PROCEDURE cs_parse_url(
  v_url    IN VARCHAR,
  v_host   OUT VARCHAR,  -- Este é passado de volta,
  v_path   OUT VARCHAR,  -- este também,
  v_query  OUT VARCHAR)  -- e este
IS
  a_pos1 INTEGER;
  a_pos2 INTEGER;
BEGIN
  v_host := NULL;
  v_path := NULL;
  v_query := NULL;
  a_pos1 := instr(v_url, '//');

  IF a_pos1 = 0 THEN
    RETURN;
  END IF;
  a_pos2 := instr(v_url, '/', a_pos1 + 2);
  IF a_pos2 = 0 THEN
    v_host := substr(v_url, a_pos1 + 2);
    v_path := '/';
    RETURN;
  END IF;

  v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
  a_pos1 := instr(v_url, '?', a_pos2 + 1);

```

```

IF a_pos1 = 0 THEN
    v_path := substr(v_url, a_pos2);
    RETURN;
END IF;

v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;

```

Abaixo está mostrada uma conversão possível para PL/pgSQL:

```

CREATE TYPE cs_parse_url_result AS (
    v_host VARCHAR,
    v_path VARCHAR,
    v_query VARCHAR
);

CREATE OR REPLACE FUNCTION cs_parse_url(v_url VARCHAR)
RETURNS cs_parse_url_result AS $$
DECLARE
    res cs_parse_url_result;
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    res.v_host := NULL;
    res.v_path := NULL;
    res.v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN res;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        res.v_host := substr(v_url, a_pos1 + 2);
        res.v_path := '/';
        RETURN res;
    END IF;

    res.v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        res.v_path := substr(v_url, a_pos2);
        RETURN res;
    END IF;

    res.v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    res.v_query := substr(v_url, a_pos1 + 1);
    RETURN res;
END;
$$ LANGUAGE plpgsql;

```

Esta função poderia ser utilizada da seguinte maneira:

```

SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');

```

```

v_host | v_path | v_query
-----+-----+-----
foobar.com | /query.cgi | baz
(1 linha)

```



O Exemplo 35-10 mostra como converter um procedimento que utiliza diversas funcionalidades específicas do Oracle.

### Exemplo 35-10. Conversão de um procedimento de PL/SQL para PL/pgSQL

A versão Oracle:

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
    PRAGMA AUTONOMOUS_TRANSACTION;❶
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;❷

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- liberar bloqueio❸
        raise_application_error(-20000,
            'Não foi possível criar uma nova tarefa: já há uma em execução.');
```

END IF;

```
DELETE FROM cs_active_job;
INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

BEGIN
    INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, sysdate);
EXCEPTION
    WHEN dup_val_on_index THEN NULL; -- não se preocupar se já existe
END;
COMMIT;
END;
/
show errors
```

Procedimentos como este podem ser facilmente convertidos em funções PostgreSQL que retornam void. Este procedimento, em particular, é interessante porque pode ensinar algumas coisas:

- ❶ Não existe a instrução PRAGMA no PostgreSQL.
- ❷ Na linguagem PL/pgSQL, após LOCK TABLE ser executado o bloqueio não é liberado até que a transação que fez a chamada termine.
- ❸ Não pode ser executada a instrução COMMIT em uma função PL/pgSQL. A função executa dentro de outra transação externa e, portanto, o COMMIT implicaria no término da execução da função. Entretanto, neste caso em particular não é preciso de qualquer forma, porque o bloqueio obtido por LOCK TABLE é liberado quando se gera um erro.

Abaixo está mostrada uma maneira de como esta função poderia ser convertida para PL/pgSQL:

```
CREATE OR REPLACE FUNCTION cs_create_job(v_job_id integer) RETURNS void AS $$
DECLARE
    a_running_job_count integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        RAISE EXCEPTION 'Não foi possível criar uma nova tarefa: já há uma em execução.';❶
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
```

```

        WHEN unique_violation THEN ❷
            -- não se preocupar se já existe
        END;

    RETURN;
END;
$$ LANGUAGE plpgsql;

```

- ❶ A sintaxe da instrução `RAISE` é bastante diferente da instrução semelhante do Oracle.
- ❷ Os nomes das exceções suportadas pelo PL/pgSQL são diferentes do Oracle. O conjunto de nomes de exceção nativos é muito maior (consulte o Apêndice A). No momento não há como declarar nomes de exceção definidos pelo usuário.

A principal diferença funcional entre este procedimento e o equivalente do Oracle é que o bloqueio exclusivo na tabela `cs_jobs` é mantido até o término da transação que fez a chamada. Também, se quem fez a chamada for posteriormente interrompido (por exemplo, devido a um erro), os efeitos deste procedimento serão desfeitos.

### 35.11.2. Outros detalhes a serem observados

Esta seção explica algumas poucas outras coisas a serem observadas ao converter funções da linguagem PL/SQL do Oracle para o PostgreSQL.

#### 35.11.2.1. Desfaz implicitamente após as exceções

Na linguagem PL/pgSQL, quando uma exceção é capturada pela cláusula `EXCEPTION` todas as alterações no banco de dados desde o começo do bloco (`BEGIN`) são desfeitas automaticamente, ou seja, este comportamento é equivalente ao que seria obtido no Oracle utilizando:

```

BEGIN
    SAVEPOINT s1;
    ... código ...
EXCEPTION
    WHEN ... THEN
        ROLLBACK TO s1;
        ... código ...
    WHEN ... THEN
        ROLLBACK TO s1;
        ... código ...
END;

```

Caso se esteja convertendo um procedimento do Oracle que utilize `SAVEPOINT` e `ROLLBACK TO` neste estilo, a tarefa é fácil: basta omitir o `SAVEPOINT` e o `ROLLBACK TO`. Se o procedimento utilizar `SAVEPOINT` e `ROLLBACK TO` de uma maneira diferente, então será necessário pensar sobre o assunto.

#### 35.11.2.2. EXECUTE

A versão PL/pgSQL do `EXECUTE` trabalha de forma semelhante à versão PL/SQL, mas é necessário lembrar de utilizar as funções `quote_literal(text)` e `quote_string(text)` conforme descrito na Seção 35.6.5. As construções do tipo `EXECUTE 'SELECT * FROM $1'`; não funcionam, a menos que estas funções sejam utilizadas.

#### 35.11.2.3. Otimização das funções PL/pgSQL

O PostgreSQL disponibiliza dois modificadores na criação da função para otimizar a execução: “volatilidade” (se a função sempre retorna o mesmo resultado quando são passados os mesmos argumentos); se é “estrita” (se a função retorna nulo se algum dos argumentos for nulo). Para obter mais detalhes deve ser consultada a página de referência do comando `CREATE FUNCTION`.

Para fazer uso dos atributos de otimização, o comando `CREATE FUNCTION` deve ficar parecido com:

```

CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

```

### 35.11.3. Apêndice

Esta seção contém o código de um conjunto de funções compatíveis com a função `instr` do Oracle que podem ser utilizadas para diminuir o esforço de conversão.

```
--
-- Funções instr equivalentes a do Oracle.
-- Sintaxe: instr(string1, string2, [n], [m])
-- onde [] indica que os parâmetros são opcionais.
--
-- Procura em string1, a partir no n-ésimo caractere, a m-ésima
-- ocorrência de string2. Se n for negativo, procura para trás.
-- Se m não for fornecido, é assumido como sendo igual a 1
-- (procura a partir do primeiro caractere).
--

CREATE FUNCTION instr(vchar, varchar) RETURNS integer AS $$
DECLARE
    pos integer;
BEGIN
    pos:= instr($1, $2, 1);
    RETURN pos;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE FUNCTION instr(string varchar, string_to_search varchar, beg_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    END IF;
END;

```

```

    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE FUNCTION instr(string varchar, string_to_search varchar,
                     beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        beg := beg_index;
        temp_str := substring(string FROM beg_index);

        FOR i IN 1..occur_index LOOP
            pos := position(string_to_search IN temp_str);

            IF i = 1 THEN
                beg := beg + pos - 1;
            ELSE
                beg := beg + pos;
            END IF;

            temp_str := substring(string FROM beg + 1);
        END LOOP;

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN beg;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                occur_number := occur_number + 1;

                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

```

## Notas

1. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
2. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
3. Exemplo escrito pelo tradutor, não fazendo parte do manual original, baseado em exemplo da lista de discussão postgresql (<http://archives.postgresql.org/postgresql-sql/2005-08/msg00198.php>).
4. Tirado da lista de discussão. (N. do T.)
5. Oracle — o Oracle utiliza as convenções: `verbo_substantivo` para procedimentos, funções e gatilhos (por exemplo, `admitir_empregado`, `obter_salario` e `auditar_empregado`, respectivamente); a convenção `v_substantivo` para as variáveis (por exemplo, `v_nome`); a convenção `cursor_substantivo` para os cursores (por exemplo, `cursor_empregado`, entre outras. Fonte: *Develop Applications Using Database Procedures 7.2*, PO1 Student Guide, Oracle. (N. do T.)

# Capítulo 36. PL/Tcl - Tcl Procedural Language

PL/Tcl is a loadable procedural language for the PostgreSQL database system that enables the Tcl (<http://www.tcl.tk/>) language to be used to write functions and trigger procedures.

## 36.1. Visão geral

PL/Tcl offers most of the capabilities a function writer has in the C language, except for some restrictions.

The good restriction is that everything is executed in a safe Tcl interpreter. In addition to the limited command set of safe Tcl, only a few commands are available to access the database via SPI and to raise messages via `elog()`. There is no way to access internals of the database server or to gain OS-level access under the permissions of the PostgreSQL server process, as a C function can do. Thus, any unprivileged database user may be permitted to use this language.

The other, implementation restriction is that Tcl functions cannot be used to create input/output functions for new data types.

Sometimes it is desirable to write Tcl functions that are not restricted to safe Tcl. For example, one might want a Tcl function that sends email. To handle these cases, there is a variant of PL/Tcl called `PL/TclU` (for untrusted Tcl). This is the exact same language except that a full Tcl interpreter is used. *If PL/TclU is used, it must be installed as an untrusted procedural language* so that only database superusers can create functions in it. The writer of a PL/TclU function must take care that the function cannot be used to do anything unwanted, since it will be able to do anything that could be done by a user logged in as the database administrator.

The shared object for the PL/Tcl and PL/TclU call handlers is automatically built and installed in the PostgreSQL library directory if Tcl support is specified in the configuration step of the installation procedure. To install PL/Tcl and/or PL/TclU in a particular database, use the `createlang` program, for example `createlang pltcl nome_do_banco_de_dados` or `createlang pltclu nome_do_banco_de_dados`.

## 36.2. PL/Tcl Functions and Arguments

To create a function in the PL/Tcl language, use the standard syntax:

```
CREATE FUNCTION nome_da_função (argument-types) RETURNS return-type AS $$
    # PL/Tcl function body
$$ LANGUAGE pltcl;
```

PL/TclU is the same, except that the language has to be specified as `pltclu`.

The body of the function is simply a piece of Tcl script. When the function is called, the argument values are passed as variables `$1 ... $n` to the Tcl script. The result is returned from the Tcl code in the usual way, with a `return` statement.

For example, a function returning the greater of two integer values could be defined as:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl STRICT;
```

Note the clause `STRICT`, which saves us from having to think about null input values: if a null value is passed, the function will not be called at all, but will just return a null result automatically.

In a nonstrict function, if the actual value of an argument is null, the corresponding `$n` variable will be set to an empty string. To detect whether a particular argument is null, use the function `argisnull`. For example, suppose that we wanted `tcl_max` with one null and one nonnull argument to return the nonnull argument, rather than null:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }

```

```

    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
}
$$ LANGUAGE pltcl;

```

As shown above, to return a null value from a PL/Tcl function, execute `return_null`. This can be done whether the function is strict or not.

Composite-type arguments are passed to the function as Tcl arrays. The element names of the array are the attribute names of the composite type. If an attribute in the passed row has the null value, it will not appear in the array. Here is an example:

```

CREATE TABLE employee (
    name text,
    salary integer,
    age integer
);

CREATE FUNCTION overpaid(employee) RETURNS boolean AS $$
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
$$ LANGUAGE pltcl;

```

There is currently no support for returning a composite-type result value, nor for returning sets.

PL/Tcl does not currently have full support for domain types: it treats a domain the same as the underlying scalar type. This means that constraints associated with the domain will not be enforced. This is not an issue for function arguments, but it is a hazard if you declare a PL/Tcl function as returning a domain type.

### 36.3. Data Values in PL/Tcl

The argument values supplied to a PL/Tcl function's code are simply the input arguments converted to text form (just as if they had been displayed by a `SELECT` statement). Conversely, the `return` command will accept any string that is acceptable input format for the function's declared return type. So, within the PL/Tcl function, all values are just text strings.

### 36.4. Global Data in PL/Tcl

Sometimes it is useful to have some global data that is held between two calls to a function or is shared between different functions. This is easily done since all PL/Tcl functions executed in one session share the same safe Tcl interpreter. So, any global Tcl variable is accessible to all PL/Tcl function calls and will persist for the duration of the SQL session. (Note that PL/TclU functions likewise share global data, but they are in a different Tcl interpreter and cannot communicate with PL/Tcl functions.)

To help protect PL/Tcl functions from unintentionally interfering with each other, a global array is made available to each function via the `upvar` command. The global name of this variable is the function's internal name, and the local name is `GD`. It is recommended that `GD` be used for persistent private data of a function. Use regular Tcl global variables only for values that you specifically intend to be shared among multiple functions.

An example of using `GD` appears in the `spi_execp` example below.

### 36.5. Database Access from PL/Tcl

The following commands are available to access the database from the body of a PL/Tcl function:

```
spi_exec ?-count n? ?-array nome? command ?loop-body?
```

Executes an SQL command given as a string. An error in the command causes an error to be raised. Otherwise, the return value of `spi_exec` is the number of rows processed (selected, inserted, updated, or deleted) by the command, or zero if the command is a utility statement. In addition, if the command is a `SELECT` statement, the values of the selected columns are placed in Tcl variables as described below.

The optional `-count` value tells `spi_exec` the maximum number of rows to process in the command. The effect of this is comparable to setting up a query as a cursor and then saying `FETCH n`.

If the command is a `SELECT` statement, the values of the result columns are placed into Tcl variables named after the columns. If the `-array` option is given, the column values are instead stored into the named associative array, with the column names used as array indexes.

If the command is a `SELECT` statement and no `loop-body` script is given, then only the first row of results are stored into Tcl variables; remaining rows, if any, are ignored. No storing occurs if the query returns no rows. (This case can be detected by checking the result of `spi_exec`.) For example,

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

will set the Tcl variable `$cnt` to the number of rows in the `pg_proc` system catalog.

If the optional `loop-body` argument is given, it is a piece of Tcl script that is executed once for each row in the query result. (`loop-body` is ignored if the given command is not a `SELECT`.) The values of the current row's columns are stored into Tcl variables before each iteration. For example,

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

will print a log message for every row of `pg_class`. This feature works similarly to other Tcl looping constructs; in particular `continue` and `break` work in the usual way inside the loop body.

If a column of a query result is null, the target variable for it is “unset” rather than being set.

```
spi_prepare comando typelist
```

Prepares and saves a query plan for later execution. The saved plan will be retained for the life of the current session.

The query may use parameters, that is, placeholders for values to be supplied whenever the plan is actually executed. In the query string, refer to parameters by the symbols `$1 ... $n`. If the query uses parameters, the names of the parameter types must be given as a Tcl list. (Write an empty list for `typelist` if no parameters are used.) Presently, the parameter types must be identified by the internal type names shown in the system table `pg_type`; for example `int4` not `integer`.

The return value from `spi_prepare` is a query ID to be used in subsequent calls to `spi_execp`. See `spi_execp` for an example.

```
spi_execp ?-count n? ?-array nome? ?-nulls string? queryid ?value-list? ?loop-body?
```

Executes a query previously prepared with `spi_prepare`. `queryid` is the ID returned by `spi_prepare`. If the query references parameters, a `value-list` must be supplied. This is a Tcl list of actual values for the parameters. The list must be the same length as the parameter type list previously given to `spi_prepare`. Omit `value-list` if the query has no parameters.

The optional value for `-nulls` is a string of spaces and 'n' characters telling `spi_execp` which of the parameters are null values. If given, it must have exactly the same length as the `value-list`. If it is not given, all the parameter values are nonnull.

Except for the way in which the query and its parameters are specified, `spi_execp` works just like `spi_exec`. The `-count`, `-array`, and `loop-body` options are the same, and so is the result value.

Here's an example of a PL/Tcl function using a prepared plan:

```
CREATE FUNCTION tl_count(integer, integer) RETURNS integer AS $$
    if {![ info exists GD(plan) ]} {
        # prepare the saved plan on the first call
        set GD(plan) [ spi_prepare \
            "SELECT count(*) AS cnt FROM tl WHERE num >= \ $1 AND num <= \ $2" \
```



```

        [ list int4 int4 ] ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
$$ LANGUAGE pltcl;

```

We need backslashes inside the query string given to `spi_prepare` to ensure that the `$n` markers will be passed through to `spi_prepare` as-is, and not replaced by Tcl variable substitution.

`spi_lastoid`

Returns the OID of the row inserted by the last `spi_exec` or `spi_execp`, if the command was a single-row `INSERT`. (If not, you get zero.)

`quote string`

Doubles all occurrences of single quote and backslash characters in the given string. This may be used to safely quote strings that are to be inserted into SQL commands given to `spi_exec` or `spi_prepare`. For example, think about an SQL command string like

```
"SELECT '$val' AS ret"
```

where the Tcl variable `val` actually contains `doesn't`. This would result in the final command string

```
SELECT 'doesn't' AS ret
```

which would cause a parse error during `spi_exec` or `spi_prepare`. To work properly, the submitted command should contain

```
SELECT 'doesn''t' AS ret
```

which can be formed in PL/Tcl using

```
"SELECT '[ quote $val ]' AS ret"
```

One advantage of `spi_execp` is that you don't have to quote parameter values like this, since the parameters are never parsed as part of an SQL command string.

`elog level msg`

Emits a log or error message. Possible levels are `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, and `FATAL`. `ERROR` raises an error condition; if this is not trapped by the surrounding Tcl code, the error propagates out to the calling query, causing the current transaction or subtransaction to be aborted. This is effectively the same as the Tcl `error` command. `FATAL` aborts the transaction and causes the current session to shut down. (There is probably no good reason to use this error level in PL/Tcl functions, but it's provided for completeness.) The other levels only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the `log_min_messages` and `client_min_messages` configuration variables. See [Seção 16.4](#) for more information.

## 36.6. Trigger Procedures in PL/Tcl

Trigger procedures can be written in PL/Tcl. PostgreSQL requires that a procedure that is to be called as a trigger must be declared as a function with no arguments and a return type of `trigger`.

The information from the trigger manager is passed to the procedure body in the following variables:

`$TG_name`

The name of the trigger from the `CREATE TRIGGER` statement.

`$TG_relid`

The object ID of the table that caused the trigger procedure to be invoked.

`$TG_relatts`

A Tcl list of the table column names, prefixed with an empty list element. So looking up a column name in the list with Tcl's `lsearch` command returns the element's number starting with 1 for the first column, the same way the columns are customarily numbered in PostgreSQL. (Empty list elements also appear in the positions of columns that have been dropped, so that the attribute numbering is correct for columns to their right.)

`$TG_when`

The string `BEFORE` or `AFTER` depending on the type of trigger call.

`$TG_level`

The string `ROW` or `STATEMENT` depending on the type of trigger call.

`$TG_op`

The string `INSERT`, `UPDATE`, or `DELETE` depending on the type of trigger call.

`$NEW`

An associative array containing the values of the new table row for `INSERT` or `UPDATE` actions, or empty for `DELETE`. The array is indexed by column name. Columns that are null will not appear in the array.

`$OLD`

An associative array containing the values of the old table row for `UPDATE` or `DELETE` actions, or empty for `INSERT`. The array is indexed by column name. Columns that are null will not appear in the array.

`$args`

A Tcl list of the arguments to the procedure as given in the `CREATE TRIGGER` statement. These arguments are also accessible as `$1 ... $n` in the procedure body.

The return value from a trigger procedure can be one of the strings `OK` or `SKIP`, or a list as returned by the `array get` Tcl command. If the return value is `OK`, the operation (`INSERT/UPDATE/DELETE`) that fired the trigger will proceed normally. `SKIP` tells the trigger manager to silently suppress the operation for this row. If a list is returned, it tells PL/Tcl to return a modified row to the trigger manager that will be inserted instead of the one given in `$NEW`. (This works for `INSERT` and `UPDATE` only.) Needless to say that all this is only meaningful when the trigger is `BEFORE` and `FOR EACH ROW`; otherwise the return value is ignored.

Here's a little example trigger procedure that forces an integer value in a table to keep track of the number of updates that are performed on the row. For new rows inserted, the value is initialized to 0 and then incremented on every update operation.

```

CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
$$ LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

Notice that the trigger procedure itself does not know the column name; that's supplied from the trigger arguments. This lets the trigger procedure be reused with different tables.

### 36.7. Modules and the `unknown` command

PL/Tcl has support for autoloading Tcl code when used. It recognizes a special table, `pltcl_modules`, which is presumed to contain modules of Tcl code. If this table exists, the module `unknown` is fetched from the table and loaded into the Tcl interpreter immediately after creating the interpreter.

While the `unknown` module could actually contain any initialization script you need, it normally defines a Tcl `unknown` procedure that is invoked whenever Tcl does not recognize an invoked procedure name. PL/Tcl's standard version of this procedure tries to find a module in `pltcl_modules` that will define the required procedure. If one is found, it is loaded into the interpreter, and then execution is allowed to proceed with the originally attempted procedure call. A secondary table `pltcl_modfuncs` provides an index of which functions are defined by which modules, so that the lookup is reasonably quick.

The PostgreSQL distribution includes support scripts to maintain these tables: `pltcl_loadmod`, `pltcl_listmod`, `pltcl_delmod`, as well as source for the standard `unknown` module in `share/unknown.pltcl`. This module must be loaded into each database initially to support the autoloading mechanism.

The tables `pltcl_modules` and `pltcl_modfuncs` must be readable by all, but it is wise to make them owned and writable only by the database administrator.

### 36.8. Tcl Procedure Names

In PostgreSQL, one and the same function name can be used for different functions as long as the number of arguments or their types differ. Tcl, however, requires all procedure names to be distinct. PL/Tcl deals with this by making the internal Tcl procedure names contain the object ID of the function from the system table `pg_proc` as part of their name. Thus, PostgreSQL functions with the same name and different argument types will be different Tcl procedures, too. This is not normally a concern for a PL/Tcl programmer, but it might be visible when debugging.

# Capítulo 37. PL/Perl - Perl Procedural Language

PL/Perl is a loadable procedural language that enables you to write PostgreSQL functions in the Perl (<http://www.perl.com>) programming language.

To install PL/Perl in a particular database, use `createlang plperl nome_do_banco_de_dados`.

**Dica:** If a language is installed into `template1`, all subsequently created databases will have the language installed automatically.

**Nota:** Users of source packages must specially enable the build of PL/Perl during the installation process. (Refer to Seção 14.1 for more information.) Users of binary packages might find PL/Perl in a separate subpackage.

## 37.1. PL/Perl Functions and Arguments

To create a function in the PL/Perl language, use the standard syntax:

```
CREATE FUNCTION nome_da_função (argument-types) RETURNS return-type AS $$
    # PL/Perl function body
$$ LANGUAGE plperl;
```

The body of the function is ordinary Perl code.

The syntax of the `CREATE FUNCTION` command requires the function body to be written as a string constant. It is usually most convenient to use dollar quoting (see Seção 4.1.2.2) for the string constant. If you choose to use regular single-quoted string constant syntax, you must escape single quote marks (`'`) and backslashes (`\`) used in the body of the function, typically by doubling them (see Seção 4.1.2.1).

Arguments and results are handled as in any other Perl subroutine: arguments are passed in `@_`, and a result value is returned with `return` or as the last expression evaluated in the function.

For example, a function returning the greater of two integer values could be defined as:

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    if ($_[0] > $_[1]) { return $_[0]; }
    return $_[1];
$$ LANGUAGE plperl;
```

If an SQL null value is passed to a function, the argument value will appear as “undefined” in Perl. The above function definition will not behave very nicely with null inputs (in fact, it will act as though they are zeroes). We could add `STRICT` to the function definition to make PostgreSQL do something more reasonable: if a null value is passed, the function will not be called at all, but will just return a null result automatically. Alternatively, we could check for undefined inputs in the function body. For example, suppose that we wanted `perl_max` with one null and one nonnull argument to return the nonnull argument, rather than a null value:

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    my ($a,$b) = @_;
    if (! defined $a) {
        if (! defined $b) { return undef; }
        return $b;
    }
    if (! defined $b) { return $a; }
    if ($a > $b) { return $a; }
    return $b;
$$ LANGUAGE plperl;
```

As shown above, to return an SQL null value from a PL/Perl function, return an undefined value. This can be done whether the function is strict or not.

Composite-type arguments are passed to the function as references to hashes. The keys of the hash are the attribute names of the composite type. Here is an example:

```
CREATE TABLE employee (
    name text,
    basesalary integer,
    bonus integer
);
```

```
CREATE FUNCTION empcomp(employee) RETURNS integer AS $$
    my ($emp) = @_;
    return $emp->{basesalary} + $emp->{bonus};
$$ LANGUAGE plperl;
```

```
SELECT name, empcomp(employee.*) FROM employee;
```

A PL/Perl function can return a composite-type result using the same approach: return a reference to a hash that has the required attributes. For example,

```
CREATE TYPE testrowperl AS (f1 integer, f2 text, f3 text);

CREATE OR REPLACE FUNCTION perl_row() RETURNS testrowperl AS $$
    return {f2 => 'hello', f1 => 1, f3 => 'world'};
$$ LANGUAGE plperl;

SELECT * FROM perl_row();
```

Any columns in the declared result data type that are not present in the hash will be returned as NULLs.

PL/Perl functions can also return sets of either scalar or composite types. To do this, return a reference to an array that contains either scalars or references to hashes, respectively. Here are some simple examples:

```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER AS $$
    return [0..$_[0]];
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);
```

```
CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testrowperl AS $$
    return [
        { f1 => 1, f2 => 'Hello', f3 => 'World' },
        { f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' },
        { f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' }
    ];
$$ LANGUAGE plperl;

SELECT * FROM perl_set();
```

Note that when you do this, Perl will have to build the entire array in memory; therefore the technique does not scale to very large result sets.

PL/Perl does not currently have full support for domain types: it treats a domain the same as the underlying scalar type. This means that constraints associated with the domain will not be enforced. This is not an issue for function arguments, but it is a hazard if you declare a PL/Perl function as returning a domain type.

## 37.2. Database Access from PL/Perl

Access to the database itself from your Perl function can be done via the function `spi_exec_query` described below, or via an experimental module `DBD::PgSPI` (<http://www.cpan.org/modules/by-module/DBD/APILOS/>) (also available at CPAN mirror sites (<http://www.cpan.org/SITES.html>)). This module makes available a DBI-compliant database-handle named `$pg_dbh` that can be used to perform queries with normal DBI syntax.

PL/Perl itself presently provides two additional Perl commands:

```
spi_exec_query(comando [, max-rows])
spi_exec_query(command)
```

Executes an SQL command. Here is an example of a query (`SELECT` command) with the optional maximum number of rows:

```
$rv = spi_exec_query('SELECT * FROM my_table', 5);
```

This returns up to 5 rows from the table `my_table`. If `my_table` has a column `my_column`, you can get that value from row `$i` of the result like this:

```
$foo = $rv->{rows}[$i]->{my_column};
```

The total number of rows returned from a `SELECT` query can be accessed like this:

```
$nrows = $rv->{processed}
```

Here is an example using a different command type:

```
$query = "INSERT INTO my_table VALUES (1, 'test')";
$rv = spi_exec_query($query);
```

You can then access the command status (e.g., `SPI_OK_INSERT`) like this:

```
$res = $rv->{status};
```

To get the number of rows affected, do:

```
$nrows = $rv->{processed};
```

Here is a complete example:

```
CREATE TABLE test (
    i int,
    v varchar
);

INSERT INTO test (i, v) VALUES (1, 'first line');
INSERT INTO test (i, v) VALUES (2, 'second line');
INSERT INTO test (i, v) VALUES (3, 'third line');
INSERT INTO test (i, v) VALUES (4, 'immortal');

CREATE FUNCTION test_munge() RETURNS SETOF test AS $$
my $res = [];
my $rv = spi_exec_query('select i, v from test;');
my $status = $rv->{status};
my $nrows = $rv->{processed};
foreach my $rn (0 .. $nrows - 1) {
    my $row = $rv->{rows}[$rn];
    $row->{i} += 200 if defined($row->{i});
    $row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
    push @$res, $row;
}
return $res;
$$ LANGUAGE plperl;

SELECT * FROM test_munge();

elog(level, msg)
```

Emit a log or error message. Possible levels are `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, and `ERROR`. `ERROR` raises an error condition; if this is not trapped by the surrounding Perl code, the error propagates out to the calling query, causing the current transaction or subtransaction to be aborted. This is effectively the same as the Perl `die` command. The other levels only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the `log_min_messages` and `client_min_messages` configuration variables. See Seção 16.4 for more information.

### 37.3. Data Values in PL/Perl

The argument values supplied to a PL/Perl function's code are simply the input arguments converted to text form (just as if they had been displayed by a `SELECT` statement). Conversely, the `return` command will accept any string that is acceptable input format for the function's declared return type. So, within the PL/Perl function, all values are just text strings.

### 37.4. Global Values in PL/Perl

You can use the global hash `$_SHARED` to store data, including code references, between function calls for the lifetime of the current session.

Here is a simple example for shared data:

```
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS text AS $$
    if ($_SHARED{$_[0]} = $_[1]) {
        return 'ok';
    } else {
        return "can't set shared variable $_[0] to $_[1]";
    }
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
    return $_SHARED{$_[0]};
$$ LANGUAGE plperl;

SELECT set_var('sample', 'Hello, PL/Perl! How's tricks?');
SELECT get_var('sample');
```

Here is a slightly more complicated example using a code reference:

```
CREATE OR REPLACE FUNCTION myfuncs() RETURNS void AS $$
    $_SHARED{myquote} = sub {
        my $arg = shift;
        $arg =~ s/(['\\])/\\$1/g;
        return "'$arg'";
    };
$$ LANGUAGE plperl;

SELECT myfuncs(); /* initializes the function */

/* Set up a function that uses the quote function */

CREATE OR REPLACE FUNCTION use_quote(TEXT) RETURNS text AS $$
    my $text_to_quote = shift;
    my $qfunc = $_SHARED{myquote};
    return &$qfunc($text_to_quote);
$$ LANGUAGE plperl;
```

(You could have replaced the above with the one-liner `return $_SHARED{myquote}->($_[0]);` at the expense of readability.)

### 37.5. Trusted and Untrusted PL/Perl

Normally, PL/Perl is installed as a “trusted” programming language named `plperl`. In this setup, certain Perl operations are disabled to preserve security. In general, the operations that are restricted are those that interact with the environment. This includes file handle operations, `require`, and `use` (for external modules). There is no way to access internals of the database server process or to gain OS-level access with the permissions of the server process, as a C function can do. Thus, any unprivileged database user may be permitted to use this language.

Here is an example of a function that will not work because file system operations are not allowed for security reasons:

```
CREATE FUNCTION badfunc() RETURNS integer AS $$
    open(TEMP, ">/tmp/badfile");
    print TEMP "Gotcha!\n";
    return 1;
$$ LANGUAGE plperl;
```

The creation of the function will succeed, but executing it will not.

Sometimes it is desirable to write Perl functions that are not restricted. For example, one might want a Perl function that sends mail. To handle these cases, PL/Perl can also be installed as an “untrusted” language (usually called PL/PerlU). In this case the full Perl language is available. If the `createlang` program is used to install the language, the language name `plperlU` will select the untrusted PL/Perl variant.

The writer of a PL/PerlU function must take care that the function cannot be used to do anything unwanted, since it will be able to do anything that could be done by a user logged in as the database administrator. Note that the database system allows only database superusers to create functions in untrusted languages.

If the above function was created by a superuser using the language `plperlU`, execution would succeed.

## 37.6. PL/Perl Triggers

PL/Perl can be used to write trigger functions. In a trigger function, the hash reference `$_TD` contains information about the current trigger event. The fields of the `$_TD` hash reference are:

`$_TD->{new}` {foo}

NEW value of column foo

`$_TD->{old}` {foo}

OLD value of column foo

`$_TD->{name}`

Name of the trigger being called

`$_TD->{event}`

Trigger event: INSERT, UPDATE, DELETE, or UNKNOWN

`$_TD->{when}`

When the trigger was called: BEFORE, AFTER, or UNKNOWN

`$_TD->{level}`

The trigger level: ROW, STATEMENT, or UNKNOWN

`$_TD->{relid}`

OID of the table on which the trigger fired

`$_TD->{relname}`

Name of the table on which the trigger fired

`$_TD->{argc}`

Number of arguments of the trigger function

`@{$_TD->{args}}`

Arguments of the trigger function. Does not exist if `$_TD->{argc}` is 0.

Triggers can return one of the following:

`return;`

Execute the statement

`"SKIP"`

Don't execute the statement



"MODIFY"

Indicates that the NEW row was modified by the trigger function

Here is an example of a trigger function, illustrating some of the above:

```
CREATE TABLE test (
    i int,
    v varchar
);

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$
    if (($TD->{new}{i} >= 100) || ($TD->{new}{i} <= 0)) {
        return "SKIP";      # skip INSERT/UPDATE command
    } elsif ($TD->{new}{v} ne "immortal") {
        $TD->{new}{v} .= "(modified by trigger)";
        return "MODIFY";    # modify row and execute INSERT/UPDATE command
    } else {
        return;              # execute INSERT/UPDATE command
    }
$$ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
    BEFORE INSERT OR UPDATE ON test
    FOR EACH ROW EXECUTE PROCEDURE valid_id();
```

## 37.7. Limitations and Missing Features

The following features are currently missing from PL/Perl, but they would make welcome contributions.

- PL/Perl functions cannot call each other directly (because they are anonymous subroutines inside Perl).
- SPI is not yet fully implemented.
- In the current implementation, if you are fetching or returning very large data sets, you should be aware that these will all go into memory.

## Capítulo 38. PL/Python - Python Procedural Language

The PL/Python procedural language allows PostgreSQL functions to be written in the Python (<http://www.python.org>) language.

To install PL/Python in a particular database, use `createlang plpythonu nome_do_banco_de_dados`.

**Dica:** If a language is installed into `template1`, all subsequently created databases will have the language installed automatically.

As of PostgreSQL 7.4, PL/Python is only available as an “untrusted” language (meaning it does not offer any way of restricting what users can do in it). It has therefore been renamed to `plpythonu`. The trusted variant `plpython` may become available again in future, if a new secure execution mechanism is developed in Python.

**Nota:** Users of source packages must specially enable the build of PL/Python during the installation process. (Refer to the installation instructions for more information.) Users of binary packages might find PL/Python in a separate subpackage.

### 38.1. PL/Python Functions

Functions in PL/Python are declared in the usual way, for example

```
CREATE FUNCTION myfunc(text) RETURNS text
AS 'return args[0]'
LANGUAGE plpythonu;
```

The Python code that is given as the body of the function definition gets transformed into a Python function. For example, the above results in

```
def __plpython_procedure_myfunc_23456():
    return args[0]
```

assuming that 23456 is the OID assigned to the function by PostgreSQL.

If you do not provide a return value, Python returns the default `None`. PL/Python translates Python's `None` into the SQL null value.

The PostgreSQL function parameters are available in the global `args` list. In the `myfunc` example, `args[0]` contains whatever was passed in as the text argument. For `myfunc2(text, integer)`, `args[0]` would contain the text argument and `args[1]` the integer argument.

The global dictionary `SD` is available to store data between function calls. This variable is private static data. The global dictionary `GD` is public data, available to all Python functions within a session. Use with care.

Each function gets its own execution environment in the Python interpreter, so that global data and function arguments from `myfunc` are not available to `myfunc2`. The exception is the data in the `GD` dictionary, as mentioned above.

### 38.2. Trigger Functions

When a function is used as a trigger, the dictionary `TD` contains trigger-related values. The trigger rows are in `TD["new"]` and/or `TD["old"]` depending on the trigger event. `TD["event"]` contains the event as a string (`INSERT`, `UPDATE`, `DELETE`, or `UNKNOWN`). `TD["when"]` contains one of `BEFORE`, `AFTER`, and `UNKNOWN`. `TD["level"]` contains one of `ROW`, `STATEMENT`, and `UNKNOWN`. `TD["name"]` contains the trigger name, and `TD["relid"]` contains the OID of the table on which the trigger occurred. If the `CREATE TRIGGER` command included arguments, they are available in `TD["args"][0]` to `TD["args"][(n-1)]`.

If `TD["when"]` is `BEFORE`, you may return `None` or `"OK"` from the Python function to indicate the row is unmodified, `"SKIP"` to abort the event, or `"MODIFY"` to indicate you've modified the row.

### 38.3. Database Access

The PL/Python language module automatically imports a Python module called `plpy`. The functions and constants in this module are available to you in the Python code as `plpy.foo`. At present `plpy` implements the functions `plpy.debug(msg)`, `plpy.log(msg)`, `plpy.info(msg)`, `plpy.notice(msg)`, `plpy.warning(msg)`, `plpy.error(msg)`, and `plpy.fatal(msg)`. `plpy.error` and `plpy.fatal` actually raise a Python exception which, if uncaught, propagates out to the calling query, causing the current transaction or subtransaction to be aborted. `raise plpy.ERROR(msg)` and `raise plpy.FATAL(msg)` are equivalent to calling `plpy.error` and `plpy.fatal`, respectively. The other functions only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the `log_min_messages` and `client_min_messages` configuration variables. See Seção 16.4 for more information.

Additionally, the `plpy` module provides two functions called `execute` and `prepare`. Calling `plpy.execute` with a query string and an optional limit argument causes that query to be run and the result to be returned in a result object. The result object emulates a list or dictionary object. The result object can be accessed by row number and column name. It has these additional methods: `nrows` which returns the number of rows returned by the query, and `status` which is the `SPI_execute()` return value. The result object can be modified.

For example,

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

returns up to 5 rows from `my_table`. If `my_table` has a column `my_column`, it would be accessed as

```
foo = rv[i]["my_column"]
```

The second function, `plpy.prepare`, prepares the execution plan for a query. It is called with a query string and a list of parameter types, if you have parameter references in the query. For example:

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name = $1", [ "text" ])
```

`text` is the type of the variable you will be passing for `$1`. After preparing a statement, you use the function `plpy.execute` to run it:

```
rv = plpy.execute(plan, [ "name" ], 5)
```

The third argument is the limit and is optional.

When you prepare a plan using the PL/Python module it is automatically saved. Read the SPI documentation (Capítulo 39) for a description of what this means. In order to make effective use of this across function calls one needs to use one of the persistent storage dictionaries `SD` or `GD` (see Seção 38.1). For example:

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
    if SD.has_key("plan"):
        plan = SD["plan"]
    else:
        plan = plpy.prepare("SELECT 1")
        SD["plan"] = plan
    # rest of function
$$ LANGUAGE plpythonu;
```

# Capítulo 39. Server Programming Interface

The *Server Programming Interface* (SPI) gives writers of user-defined C functions the ability to run SQL commands inside their functions. SPI is a set of interface functions to simplify access to the parser, planner, optimizer, and executor. SPI also does some memory management.

**Nota:** The available procedural languages provide various means to execute SQL commands from procedures. Most of these facilities are based on SPI, so this documentation might be of use for users of those languages as well.

To avoid misunderstanding we'll use the term “function” when we speak of SPI interface functions and “procedure” for a user-defined C-function that is using SPI.

Note that if a command invoked via SPI fails, then control will not be returned to your procedure. Rather, the transaction or subtransaction in which your procedure executes will be rolled back. (This may seem surprising given that the SPI functions mostly have documented error-return conventions. Those conventions only apply for errors detected within the SPI functions themselves, however.) It is possible to recover control after an error by establishing your own subtransaction surrounding SPI calls that might fail. This is not currently documented because the mechanisms required are still in flux.

SPI functions return a nonnegative result on success (either via a returned integer value or in the global variable `SPI_result`, as described below). On error, a negative result or `NULL` will be returned.

Source code files that use SPI must include the header file `executor/spi.h`.

## 39.1. Interface Functions

# SPI\_connect

## Nome

`SPI_connect` — connect a procedure to the SPI manager

## Sinopse

```
int SPI_connect(void)
```

## Descrição

`SPI_connect` opens a connection from a procedure invocation to the SPI manager. You must call this function if you want to execute commands through SPI. Some utility SPI functions may be called from unconnected procedures.

If your procedure is already connected, `SPI_connect` will return the error code `SPI_ERROR_CONNECT`. This could happen if a procedure that has called `SPI_connect` directly calls another procedure that calls `SPI_connect`. While recursive calls to the SPI manager are permitted when an SQL command called through SPI invokes another function that uses SPI, directly nested calls to `SPI_connect` and `SPI_finish` are forbidden. (But see `SPI_push` and `SPI_pop`.)

## Valor retornado

`SPI_OK_CONNECT`

on success

`SPI_ERROR_CONNECT`

on error

# SPI\_finish

## Nome

`SPI_finish` — disconnect a procedure from the SPI manager

## Sinopse

```
int SPI_finish(void)
```

## Descrição

`SPI_finish` closes an existing connection to the SPI manager. You must call this function after completing the SPI operations needed during your procedure's current invocation. You do not need to worry about making this happen, however, if you abort the transaction via `elog(ERROR)`. In that case SPI will clean itself up automatically.

If `SPI_finish` is called without having a valid connection, it will return `SPI_ERROR_UNCONNECTED`. There is no fundamental problem with this; it means that the SPI manager has nothing to do.

## Valor retornado

`SPI_OK_FINISH`

if properly disconnected

`SPI_ERROR_UNCONNECTED`

if called from an unconnected procedure

# SPI\_push

## Nome

`SPI_push` — push SPI stack to allow recursive SPI usage

## Sinopse

```
void SPI_push(void)
```

## Descrição

`SPI_push` should be called before executing another procedure that might itself wish to use SPI. After `SPI_push`, SPI is no longer in a “connected” state, and SPI function calls will be rejected unless a fresh `SPI_connect` is done. This ensures a clean separation between your procedure's SPI state and that of another procedure you call. After the other procedure returns, call `SPI_pop` to restore access to your own SPI state.

Note that `SPI_execute` and related functions automatically do the equivalent of `SPI_push` before passing control back to the SQL execution engine, so it is not necessary for you to worry about this when using those functions. Only when you are directly calling arbitrary code that might contain `SPI_connect` calls do you need to issue `SPI_push` and `SPI_pop`.

# SPI\_pop

## Nome

SPI\_pop — pop SPI stack to return from recursive SPI usage

## Sinopse

```
void SPI_pop(void)
```

## Descrição

SPI\_pop pops the previous environment from the SPI call stack. See SPI\_push.



# SPI\_execute

## Nome

SPI\_execute — execute a command

## Sinopse

```
int SPI_execute(const char * command, bool read_only, int count)
```

## Descrição

SPI\_execute executes the specified SQL command for `count` rows. If `read_only` is true, the command must be read-only, and execution overhead is somewhat reduced.

This function may only be called from a connected procedure.

If `count` is zero then the command is executed for all rows that it applies to. If `count` is greater than 0, then the number of rows for which the command will be executed is restricted (much like a `LIMIT` clause). For example,

```
SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);
```

will allow at most 5 rows to be inserted into the table.

You may pass multiple commands in one string. SPI\_execute returns the result for the command executed last. The `count` limit applies to each command separately, but it is not applied to hidden commands generated by rules.

When `read_only` is false, SPI\_execute increments the command counter and computes a new *snapshot* before executing each command in the string. The snapshot does not actually change if the current transaction isolation level is `SERIALIZABLE`, but in `READ COMMITTED` mode the snapshot update allows each command to see the results of newly committed transactions from other sessions. This is essential for consistent behavior when the commands are modifying the database.

When `read_only` is true, SPI\_execute does not update either the snapshot or the command counter, and it allows only plain `SELECT` commands to appear in the command string. The commands are executed using the snapshot previously established for the surrounding query. This execution mode is somewhat faster than the read/write mode due to eliminating per-command overhead. It also allows genuinely *stable* functions to be built: since successive executions will all use the same snapshot, there will be no change in the results.

It is generally unwise to mix read-only and read-write commands within a single function using SPI; that could result in very confusing behavior, since the read-only queries would not see the results of any database updates done by the read-write queries.

The actual number of rows for which the (last) command was executed is returned in the global variable `SPI_processed` (unless the return value of the function is `SPI_OK_UTILITY`). If the return value of the function is `SPI_OK_SELECT` then you may use the global pointer `SPITupleTable *SPI_tuptable` to access the result rows.

The structure `SPITupleTable` is defined thus:

```
typedef struct
{
    MemoryContext tuptabcxt;    /* memory context of result table */
    uint32        allocated;    /* number of allocated vals */
    uint32        free;        /* number of free vals */
    TupleDesc     tupdesc;      /* row descriptor */
    HeapTuple     *vals;        /* rows */
} SPITupleTable;
```

`vals` is an array of pointers to rows. (The number of valid entries is given by `SPI_processed`.) `tupdesc` is a row descriptor which you may pass to SPI functions dealing with rows. `tuptabcxt`, `allocated`, and `free` are internal fields not intended for use by SPI callers.

`SPI_finish` frees all `SPITupleTables` allocated during the current procedure. You can free a particular result table earlier, if you are done with it, by calling `SPI_freetuptable`.

## Argumentos

`const char * command`

string containing command to execute

`bool read_only`

true for read-only execution

`int count`

maximum number of rows to process or return

## Valor retornado

If the execution of the command was successful then one of the following (nonnegative) values will be returned:

`SPI_OK_SELECT`

if a `SELECT` (but not `SELECT INTO`) was executed

`SPI_OK_SELINTO`

if a `SELECT INTO` was executed

`SPI_OK_DELETE`

if a `DELETE` was executed

`SPI_OK_INSERT`

if an `INSERT` was executed

`SPI_OK_UPDATE`

if an `UPDATE` was executed

`SPI_OK_UTILITY`

if a utility command (e.g., `CREATE TABLE`) was executed

On error, one of the following negative values is returned:

`SPI_ERROR_ARGUMENT`

if command is `NULL` or count is less than 0

`SPI_ERROR_COPY`

if `COPY TO stdout` or `COPY FROM stdin` was attempted

`SPI_ERROR_CURSOR`

if `DECLARE`, `CLOSE`, or `FETCH` was attempted

`SPI_ERROR_TRANSACTION`

if `BEGIN`, `COMMIT`, or `ROLLBACK` was attempted

`SPI_ERROR_OPUNKNOWN`

if the command type is unknown (shouldn't happen)

`SPI_ERROR_UNCONNECTED`

if called from an unconnected procedure

**Observações**

The functions `SPI_execute`, `SPI_exec`, `SPI_execute_plan`, and `SPI_execp` change both `SPI_processed` and `SPI_tuptable` (just the pointer, not the contents of the structure). Save these two global variables into local procedure variables if you need to access the result table of `SPI_execute` or a related function across later calls.

# SPI\_exec

## Nome

`SPI_exec` — execute a read/write command

## Sinopse

```
int SPI_exec(const char * command, int count)
```

## Descrição

`SPI_exec` is the same as `SPI_execute`, with the latter's `read_only` parameter always taken as `false`.

## Argumentos

`const char * command`

string containing command to execute

`int count`

maximum number of rows to process or return

## Valor retornado

See `SPI_execute`.

# SPI\_prepare

## Nome

`SPI_prepare` — prepare a plan for a command, without executing it yet

## Sinopse

```
void * SPI_prepare(const char * command, int nargs, Oid * argtypes)
```

## Descrição

`SPI_prepare` creates and returns an execution plan for the specified command but doesn't execute the command. This function should only be called from a connected procedure.

When the same or a similar command is to be executed repeatedly, it may be advantageous to perform the planning only once. `SPI_prepare` converts a command string into an execution plan that can be executed repeatedly using `SPI_execute_plan`.

A prepared command can be generalized by writing parameters (\$1, \$2, etc.) in place of what would be constants in a normal command. The actual values of the parameters are then specified when `SPI_execute_plan` is called. This allows the prepared command to be used over a wider range of situations than would be possible without parameters.

The plan returned by `SPI_prepare` can be used only in the current invocation of the procedure, since `SPI_finish` frees memory allocated for a plan. But a plan can be saved for longer using the function `SPI_saveplan`.

## Argumentos

`const char * command`

command string

`int nargs`

number of input parameters (\$1, \$2, etc.)

`Oid * argtypes`

pointer to an array containing the OIDs of the data types of the parameters

## Valor retornado

`SPI_prepare` returns a non-null pointer to an execution plan. On error, `NULL` will be returned, and `SPI_result` will be set to one of the same error codes used by `SPI_execute`, except that it is set to `SPI_ERROR_ARGUMENT` if `command` is `NULL`, or if `nargs` is less than 0, or if `nargs` is greater than 0 and `argtypes` is `NULL`.

## Observações

There is a disadvantage to using parameters: since the planner does not know the values that will be supplied for the parameters, it may make worse planning choices than it would make for a normal command with all constants visible.

# SPI\_getargcount

## Nome

`SPI_getargcount` — return the number of arguments needed by a plan prepared by `SPI_prepare`

## Sinopse

```
int SPI_getargcount(void * plan)
```

## Descrição

`SPI_getargcount` returns the number of arguments needed to execute a plan prepared by `SPI_prepare`.

## Argumentos

```
void * plan  
    execution plan (returned by SPI_prepare)
```

## Valor retornado

The expected argument count for the `plan`, or `SPI_ERROR_ARGUMENT` if the `plan` is `NULL`

# SPI\_getargtypeid

## Nome

`SPI_getargtypeid` — return the data type OID for an argument of a plan prepared by `SPI_prepare`

## Sinopse

```
Oid SPI_getargtypeid(void * plan, int argIndex)
```

## Descrição

`SPI_getargtypeid` returns the OID representing the type id for the `argIndex`'th argument of a plan prepared by `SPI_prepare`. First argument is at index zero.

## Argumentos

```
void * plan
    execution plan (returned by SPI_prepare)

int argIndex
    zero based index of the argument
```

## Valor retornado

The type id of the argument at the given index, or `SPI_ERROR_ARGUMENT` if the `plan` is `NULL` or `argIndex` is less than 0 or not less than the number of arguments declared for the `plan`

# SPI\_is\_cursor\_plan

## Nome

`SPI_is_cursor_plan` — return true if a plan prepared by `SPI_prepare` can be used with `SPI_cursor_open`

## Sinopse

```
bool SPI_is_cursor_plan(void * plan)
```

## Descrição

`SPI_is_cursor_plan` returns true if a plan prepared by `SPI_prepare` can be passed as an argument to `SPI_cursor_open` and false if that is not the case. The criteria are that the plan represents one single command and that this command is a `SELECT` without an `INTO` clause.

## Argumentos

```
void * plan  
    execution plan (returned by SPI_prepare)
```

## Valor retornado

true or false to indicate if the plan can produce a cursor or not, or `SPI_ERROR_ARGUMENT` if the plan is NULL



# SPI\_execute\_plan

## Nome

SPI\_execute\_plan — execute a plan prepared by SPI\_prepare

## Sinopse

```
int SPI_execute_plan(void * plan, Datum * values, const char * nulls,
                    bool read_only, int count)
```

## Descrição

SPI\_execute\_plan executes a plan prepared by SPI\_prepare. read\_only and count have the same interpretation as in SPI\_execute.

## Argumentos

void \* plan

execution plan (returned by SPI\_prepare)

Datum \* values

An array of actual parameter values. Must have same length as the plan's number of arguments.

const char \* nulls

An array describing which parameters are null. Must have same length as the plan's number of arguments. n indicates a null value (entry in values will be ignored); a space indicates a nonnull value (entry in values is valid).

If nulls is NULL then SPI\_execute\_plan assumes that no parameters are null.

bool read\_only

true for read-only execution

int count

maximum number of rows to process or return

## Valor retornado

The return value is the same as for SPI\_execute, with the following additional possible error (negative) results:

SPI\_ERROR\_ARGUMENT

if plan is NULL or count is less than 0

SPI\_ERROR\_PARAM

if values is NULL and plan was prepared with some parameters

SPI\_processed and SPI\_tuptable are set as in SPI\_execute if successful.

## Observações

If one of the objects (a table, function, etc.) referenced by the prepared plan is dropped during the session then the result of SPI\_execute\_plan for this plan will be unpredictable.

# SPI\_execp

## Nome

SPI\_execp — execute a plan in read/write mode

## Sinopse

```
int SPI_execp(void * plan, Datum * values, const char * nulls, int count)
```

## Descrição

SPI\_execp is the same as SPI\_execute\_plan, with the latter's read\_only parameter always taken as false.

## Argumentos

void \* plan

execution plan (returned by SPI\_prepare)

Datum \* values

An array of actual parameter values. Must have same length as the plan's number of arguments.

const char \* nulls

An array describing which parameters are null. Must have same length as the plan's number of arguments. n indicates a null value (entry in values will be ignored); a space indicates a nonnull value (entry in values is valid).

If nulls is NULL then SPI\_execp assumes that no parameters are null.

int count

maximum number of rows to process or return

## Valor retornado

See SPI\_execute\_plan.

SPI\_processed and SPI\_tuptable are set as in SPI\_execute if successful.

# SPI\_cursor\_open

## Nome

`SPI_cursor_open` — set up a cursor using a plan created with `SPI_prepare`

## Sinopse

```
Portal SPI_cursor_open(const char * name, void * plan,
                      Datum * values, const char * nulls,
                      bool read_only)
```

## Descrição

`SPI_cursor_open` sets up a cursor (internally, a portal) that will execute a plan prepared by `SPI_prepare`. The parameters have the same meanings as the corresponding parameters to `SPI_execute_plan`.

Using a cursor instead of executing the plan directly has two benefits. First, the result rows can be retrieved a few at a time, avoiding memory overrun for queries that return many rows. Second, a portal can outlive the current procedure (it can, in fact, live to the end of the current transaction). Returning the portal name to the procedure's caller provides a way of returning a row set as result.

## Argumentos

`const char * name`

name for portal, or `NULL` to let the system select a name

`void * plan`

execution plan (returned by `SPI_prepare`)

`Datum * values`

An array of actual parameter values. Must have same length as the plan's number of arguments.

`const char * nulls`

An array describing which parameters are null. Must have same length as the plan's number of arguments. `n` indicates a null value (entry in `values` will be ignored); a space indicates a nonnull value (entry in `values` is valid).

If `nulls` is `NULL` then `SPI_cursor_open` assumes that no parameters are null.

`bool read_only`

true for read-only execution

## Valor retornado

pointer to portal containing the cursor, or `NULL` on error

# SPI\_cursor\_find

## Nome

SPI\_cursor\_find — find an existing cursor by name

## Sinopse

```
Portal SPI_cursor_find(const char * name)
```

## Descrição

SPI\_cursor\_find finds an existing portal by name. This is primarily useful to resolve a cursor name returned as text by some other function.

## Argumentos

```
const char * name  
    name of the portal
```

## Valor retornado

pointer to the portal with the specified name, or NULL if none was found

# SPI\_cursor\_fetch

## Nome

`SPI_cursor_fetch` — fetch some rows from a cursor

## Sinopse

```
void SPI_cursor_fetch(Portal portal, bool forward, int count)
```

## Descrição

`SPI_cursor_fetch` fetches some rows from a cursor. This is equivalent to the SQL command `FETCH`.

## Argumentos

`Portal portal`

portal containing the cursor

`bool forward`

true for fetch forward, false for fetch backward

`int count`

maximum number of rows to fetch

## Valor retornado

`SPI_processed` and `SPI_tuptable` are set as in `SPI_execute` if successful.

# SPI\_cursor\_move

## Nome

SPI\_cursor\_move — move a cursor

## Sinopse

```
void SPI_cursor_move(Portal portal, bool forward, int count)
```

## Descrição

SPI\_cursor\_move skips over some number of rows in a cursor. This is equivalent to the SQL command `MOVE`.

## Argumentos

Portal portal

portal containing the cursor

bool forward

true for move forward, false for move backward

int count

maximum number of rows to move

# SPI\_cursor\_close

## Nome

SPI\_cursor\_close — close a cursor

## Sinopse

```
void SPI_cursor_close(Portal portal)
```

## Descrição

SPI\_cursor\_close closes a previously created cursor and releases its portal storage.

All open cursors are closed automatically at the end of a transaction. SPI\_cursor\_close need only be invoked if it is desirable to release resources sooner.

## Argumentos

Portal portal

portal containing the cursor

# SPI\_saveplan

## Nome

SPI\_saveplan — save a plan

## Sinopse

```
void * SPI_saveplan(void * plan)
```

## Descrição

SPI\_saveplan saves a passed plan (prepared by SPI\_prepare) in memory protected from freeing by SPI\_finish and by the transaction manager and returns a pointer to the saved plan. This gives you the ability to reuse prepared plans in the subsequent invocations of your procedure in the current session. You may save the pointer returned in a local variable. Always check if this pointer is NULL or not either when preparing a plan or using an already prepared plan in SPI\_execute\_plan.

## Argumentos

```
void * plan
    the plan to be saved
```

## Valor retornado

Pointer to the saved plan; NULL if unsuccessful. On error, SPI\_result is set thus:

```
SPI_ERROR_ARGUMENT
```

if plan is NULL

```
SPI_ERROR_UNCONNECTED
```

if called from an unconnected procedure

## Observações

If one of the objects (a table, function, etc.) referenced by the prepared plan is dropped during the session then the results of SPI\_execute\_plan for this plan will be unpredictable.

## 39.2. Interface Support Functions

The functions described here provide an interface for extracting information from result sets returned by SPI\_execute and other SPI functions.

All functions described in this section may be used by both connected and unconnected procedures.



# SPI\_fname

## Nome

`SPI_fname` — determine the column name for the specified column number

## Sinopse

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

## Descrição

`SPI_fname` returns a copy of the column name of the specified column. (You can use `pfree` to release the copy of the name when you don't need it anymore.)

## Argumentos

`TupleDesc rowdesc`

input row description

`int colnumber`

column number (count starts at 1)

## Valor retornado

The column name; NULL if `colnumber` is out of range. `SPI_result` set to `SPI_ERROR_NOATTRIBUTE` on error.

# SPI\_fnumber

## Nome

`SPI_fnumber` — determine the column number for the specified column name

## Sinopse

```
int SPI_fnumber(TupleDesc rowdesc, const char * colname)
```

## Descrição

`SPI_fnumber` returns the column number for the column with the specified name.

If `colname` refers to a system column (e.g., `oid`) then the appropriate negative column number will be returned. The caller should be careful to test the return value for exact equality to `SPI_ERROR_NOATTRIBUTE` to detect an error; testing the result for less than or equal to 0 is not correct unless system columns should be rejected.

## Argumentos

`TupleDesc rowdesc`

input row description

`const char * colname`

column name

## Valor retornado

Column number (count starts at 1), or `SPI_ERROR_NOATTRIBUTE` if the named column was not found.

# SPI\_getvalue

## Nome

`SPI_getvalue` — return the string value of the specified column

## Sinopse

```
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)
```

## Descrição

`SPI_getvalue` returns the string representation of the value of the specified column.

The result is returned in memory allocated using `palloc`. (You can use `pfree` to release the memory when you don't need it anymore.)

## Argumentos

`HeapTuple row`

input row to be examined

`TupleDesc rowdesc`

input row description

`int colnumber`

column number (count starts at 1)

## Valor retornado

Column value, or `NULL` if the column is null, `colnumber` is out of range (`SPI_result` is set to `SPI_ERROR_NOATTRIBUTE`), or no output function available (`SPI_result` is set to `SPI_ERROR_NOOUTFUNC`).

# SPI\_getbinval

## Nome

`SPI_getbinval` — return the binary value of the specified column

## Sinopse

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber, bool * isnull)
```

## Descrição

`SPI_getbinval` returns the value of the specified column in the internal form (as type `Datum`).

This function does not allocate new space for the datum. In the case of a pass-by-reference data type, the return value will be a pointer into the passed row.

## Argumentos

`HeapTuple row`

input row to be examined

`TupleDesc rowdesc`

input row description

`int rownumber`

column number (count starts at 1)

`bool * isnull`

flag for a null value in the column

## Valor retornado

The binary value of the column is returned. The variable pointed to by `isnull` is set to true if the column is null, else to false.

`SPI_result` is set to `SPI_ERROR_NOATTRIBUTE` on error.

# SPI\_gettype

## Nome

`SPI_gettype` — return the data type name of the specified column

## Sinopse

```
char * SPI_gettype(TupleDesc rowdesc, int colnumber)
```

## Descrição

`SPI_gettype` returns a copy of the data type name of the specified column. (You can use `pfree` to release the copy of the name when you don't need it anymore.)

## Argumentos

`TupleDesc rowdesc`

input row description

`int colnumber`

column number (count starts at 1)

## Valor retornado

The data type name of the specified column, or `NULL` on error. `SPI_result` is set to `SPI_ERROR_NOATTRIBUTE` on error.

# SPI\_gettypeid

## Nome

`SPI_gettypeid` — return the data type OID of the specified column

## Sinopse

```
Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)
```

## Descrição

`SPI_gettypeid` returns the OID of the data type of the specified column.

## Argumentos

`TupleDesc rowdesc`

input row description

`int colnumber`

column number (count starts at 1)

## Valor retornado

The OID of the data type of the specified column or `InvalidOid` on error. On error, `SPI_result` is set to `SPI_ERROR_NOATTRIBUTE`.

# SPI\_getrelname

## Nome

`SPI_getrelname` — return the name of the specified relation

## Sinopse

```
char * SPI_getrelname(Relation rel)
```

## Descrição

`SPI_getrelname` returns a copy of the name of the specified relation. (You can use `pfree` to release the copy of the name when you don't need it anymore.)

## Argumentos

`Relation rel`  
input relation

## Valor retornado

The name of the specified relation.

## 39.3. Memory Management

PostgreSQL allocates memory within *memory contexts*, which provide a convenient method of managing allocations made in many different places that need to live for differing amounts of time. Destroying a context releases all the memory that was allocated in it. Thus, it is not necessary to keep track of individual objects to avoid memory leaks; instead only a relatively small number of contexts have to be managed. `palloc` and related functions allocate memory from the “current” context.

`SPI_connect` creates a new memory context and makes it current. `SPI_finish` restores the previous current memory context and destroys the context created by `SPI_connect`. These actions ensure that transient memory allocations made inside your procedure are reclaimed at procedure exit, avoiding memory leakage.

However, if your procedure needs to return an object in allocated memory (such as a value of a pass-by-reference data type), you cannot allocate that memory using `palloc`, at least not while you are connected to SPI. If you try, the object will be deallocated by `SPI_finish`, and your procedure will not work reliably. To solve this problem, use `SPI_palloc` to allocate memory for your return object. `SPI_palloc` allocates memory in the “upper executor context”, that is, the memory context that was current when `SPI_connect` was called, which is precisely the right context for a value returned from your procedure.

If `SPI_palloc` is called while the procedure is not connected to SPI, then it acts the same as a normal `palloc`. Before a procedure connects to the SPI manager, the current memory context is the upper executor context, so all allocations made by the procedure via `palloc` or by SPI utility functions are made in this context.

When `SPI_connect` is called, the private context of the procedure, which is created by `SPI_connect`, is made the current context. All allocations made by `palloc`, `repalloc`, or SPI utility functions (except for `SPI_copytuple`, `SPI_returntuple`, `SPI_modifytuple`, and `SPI_palloc`) are made in this context. When a procedure disconnects from the SPI manager (via `SPI_finish`) the current context is restored to the upper executor context, and all allocations made in the procedure memory context are freed and cannot be used any more.

All functions described in this section may be used by both connected and unconnected procedures. In an unconnected procedure, they act the same as the underlying ordinary server functions (`palloc`, etc.).

# SPI\_palloc

## Nome

`SPI_palloc` — allocate memory in the upper executor context

## Sinopse

```
void * SPI_palloc(Size size)
```

## Descrição

`SPI_palloc` allocates memory in the upper executor context.

## Argumentos

`Size size`

size in bytes of storage to allocate

## Valor retornado

pointer to new storage space of the specified size



# SPI\_realloc

## Nome

`SPI_realloc` — reallocate memory in the upper executor context

## Sinopse

```
void * SPI_realloc(void * pointer, Size size)
```

## Descrição

`SPI_realloc` changes the size of a memory segment previously allocated using `SPI_palloc`.

This function is no longer different from plain `realloc`. It's kept just for backward compatibility of existing code.

## Argumentos

`void * pointer`

pointer to existing storage to change

`Size size`

size in bytes of storage to allocate

## Valor retornado

pointer to new storage space of specified size with the contents copied from the existing area

# SPI\_pfree

## Nome

SPI\_pfree — free memory in the upper executor context

## Sinopse

```
void SPI_pfree(void * pointer)
```

## Descrição

SPI\_pfree frees memory previously allocated using SPI\_palloc or SPI\_realloc.

This function is no longer different from plain pfree. It's kept just for backward compatibility of existing code.

## Argumentos

```
void * pointer
```

pointer to existing storage to free

# SPI\_copytuple

## Nome

`SPI_copytuple` — make a copy of a row in the upper executor context

## Sinopse

```
HeapTuple SPI_copytuple(HeapTuple row)
```

## Descrição

`SPI_copytuple` makes a copy of a row in the upper executor context. This is normally used to return a modified row from a trigger. In a function declared to return a composite type, use `SPI_returntuple` instead.

## Argumentos

`HeapTuple row`

row to be copied

## Valor retornado

the copied row; `NULL` only if `tuple` is `NULL`

# SPI\_returntuple

## Nome

`SPI_returntuple` — prepare to return a tuple as a Datum

## Sinopse

```
HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)
```

## Descrição

`SPI_returntuple` makes a copy of a row in the upper executor context, returning it in the form of a row type Datum. The returned pointer need only be converted to Datum via `PointerGetDatum` before returning.

Note that this should be used for functions that are declared to return composite types. It is not used for triggers; use `SPI_copytuple` for returning a modified row in a trigger.

## Argumentos

`HeapTuple row`

row to be copied

`TupleDesc rowdesc`

descriptor for row (pass the same descriptor each time for most effective caching)

## Valor retornado

`HeapTupleHeader` pointing to copied row; NULL only if row or rowdesc is NULL

# SPI\_modifytuple

## Nome

`SPI_modifytuple` — create a row by replacing selected fields of a given row

## Sinopse

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, ncols, colnum, Datum * values, const
char * nulls)
```

## Descrição

`SPI_modifytuple` creates a new row by substituting new values for selected columns, copying the original row's columns at other positions. The input row is not modified.

## Argumentos

`Relation rel`

Used only as the source of the row descriptor for the row. (Passing a relation rather than a row descriptor is a misfeature.)

`HeapTuple row`

row to be modified

`int ncols`

number of column numbers in the array `colnum`

`int * colnum`

array of the numbers of the columns that are to be changed (column numbers start at 1)

`Datum * values`

new values for the specified columns

`const char * Nulls`

which new values are null, if any (see `SPI_execute_plan` for the format)

## Valor retornado

new row with modifications, allocated in the upper executor context; NULL only if `row` is NULL

On error, `SPI_result` is set as follows:

`SPI_ERROR_ARGUMENT`

if `rel` is NULL, or if `row` is NULL, or if `ncols` is less than or equal to 0, or if `colnum` is NULL, or if `values` is NULL.

`SPI_ERROR_NOATTRIBUTE`

if `colnum` contains an invalid column number (less than or equal to 0 or greater than the number of column in `row`)

# SPI\_freetuple

## Nome

`SPI_freetuple` — free a row allocated in the upper executor context

## Sinopse

```
void SPI_freetuple(HeapTuple row)
```

## Descrição

`SPI_freetuple` frees a row previously allocated in the upper executor context.

This function is no longer different from plain `heap_freetuple`. It's kept just for backward compatibility of existing code.

## Argumentos

`HeapTuple row`

row to free

# SPI\_freetuptable

## Nome

`SPI_freetuptable` — free a row set created by `SPI_execute` or a similar function

## Sinopse

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

## Descrição

`SPI_freetuptable` frees a row set created by a prior SPI command execution function, such as `SPI_execute`. Therefore, this function is usually called with the global variable `SPI_tupletable` as argument.

This function is useful if a SPI procedure needs to execute multiple commands and does not want to keep the results of earlier commands around until it ends. Note that any unfreed row sets will be freed anyway at `SPI_finish`.

## Argumentos

`SPITupleTable * tuptable`

pointer to row set to free

# SPI\_freeplan

## Nome

`SPI_freeplan` — free a previously saved plan

## Sinopse

```
int SPI_freeplan(void *plan)
```

## Descrição

`SPI_freeplan` releases a command execution plan previously returned by `SPI_prepare` or saved by `SPI_saveplan`.

## Argumentos

```
void * plan
```

pointer to plan to free

## Valor retornado

`SPI_ERROR_ARGUMENT` if `plan` is `NULL`.

## 39.4. Visibility of Data Changes

The following rules govern the visibility of data changes in functions that use SPI (or any other C function):

- During the execution of an SQL command, any data changes made by the command are invisible to the command itself. For example, in  

```
INSERT INTO a SELECT * FROM a;
```

the inserted rows are invisible to the `SELECT` part.
- Changes made by a command `C` are visible to all commands that are started after `C`, no matter whether they are started inside `C` (during the execution of `C`) or after `C` is done.
- Commands executed via SPI inside a function called by an SQL command (either an ordinary function or a trigger) follow one or the other of the above rules depending on the read/write flag passed to SPI. Commands executed in read-only mode follow the first rule: they can't see changes of the calling command. Commands executed in read-write mode follow the second rule: they can see all changes made so far.
- All standard procedural languages set the SPI read-write mode depending on the volatility attribute of the function. Commands of `STABLE` and `IMMUTABLE` functions are done in read-only mode, while commands of `VOLATILE` functions are done in read-write mode. While authors of C functions are able to violate this convention, it's unlikely to be a good idea to do so.

The next section contains an example that illustrates the application of these rules.

## 39.5. Exemplos

This section contains a very simple example of SPI usage. The procedure `execq` takes an SQL command as its first argument and a row count as its second, executes the command using `SPI_exec` and returns the number of rows that were processed by the command. You can find more complex examples for SPI in the source tree in `src/test/regress/regress.c` and in `contrib/spi`.

```
#include "executor/spi.h"

int execq(text *sql, int cnt);
```



```

int
execq(text *sql, int cnt)
{
    char *command;
    int ret;
    int proc;

    /* Convert given text object to a C string */
    command = DatumGetCString(DirectFunctionCall1(textout,
                                                    PointerGetDatum(sql)));

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;
    /*
     * If this is a SELECT and some rows were fetched,
     * then the rows are printed via elog(INFO).
     */
    if (ret == SPI_OK_SELECT && SPI_processed > 0)
    {
        TupleDesc tupdesc = SPI_tuptable->tupdesc;
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        int i, j;

        for (j = 0; j < proc; j++)
        {
            HeapTuple tuple = tuptable->vals[j];

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen(buf), sizeof(buf) - strlen(buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog (INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(command);

    return (proc);
}

```

(This function uses call convention version 0, to make the example easier to understand. In real applications you should use the new version 1 interface.)

This is how you declare the function after having compiled it into a shared library:

```

CREATE FUNCTION execq(text, integer) RETURNS integer
AS 'nome_do_arquivo'
LANGUAGE C;

```

Here is a sample session:

```

=> SELECT execq('CREATE TABLE a (x integer)', 0);
   execq
-----
      0
(1 row)

```

```

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 167631 1
=> SELECT execq('SELECT * FROM a', 0);
INFO: EXECQ: 0      -- inserted by execq
INFO: EXECQ: 1      -- returned by execq and inserted by upper INSERT

    execq
-----
         2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
    execq
-----
         1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO: EXECQ: 0
INFO: EXECQ: 1
INFO: EXECQ: 2      -- 0 + 2, only one row inserted - as specified

    execq
-----
         3          -- 10 is the max value only, 3 is the real number of rows
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 167712 1
=> SELECT * FROM a;
    x
---
    1          -- no rows in a (0) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO: EXECQ: 0
INSERT 167713 1
=> SELECT * FROM a;
    x
---
    1
    2          -- there was one row in a + 1
(2 rows)

-- This demonstrates the data changes visibility rule:

=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO: EXECQ: 1
INFO: EXECQ: 2
INFO: EXECQ: 1
INFO: EXECQ: 2
INFO: EXECQ: 2
INSERT 0 2
=> SELECT * FROM a;
    x
---
    1
    2

```

```
2          -- 2 rows * 1 (x in first row)
6          -- 3 rows (2 + 1 just inserted) * 2 (x in second row)
(4 rows)    ^^^^^^
             rows visible to execq() in different invocations
```

# VI. Referência

As entradas nesta Referência se propõem a fornecer, em um espaço razoável, um sumário confiável, completo e formal dos respectivos assuntos. Mais informações sobre a utilização do PostgreSQL sob forma narrativa, de tutorial, ou de exemplos, podem ser encontradas em outras partes desta documentação. Consulte as referências cruzadas presentes em cada página de referência.

As entradas de referência também estão disponíveis no formato tradicional “man pages” do Unix.

# I. Comandos SQL

Esta parte contém informação de referência para os comandos SQL suportados pelo PostgreSQL. Como “SQL” a linguagem é referida de forma geral; informações sobre a conformidade e compatibilidade de cada comando com relação ao padrão podem ser encontradas nas respectivas páginas de referência.

# ABORT

## Nome

ABORT — interrompe a transação corrente

## Sinopse

```
ABORT [ WORK | TRANSACTION ]
```

## Descrição

O comando `ABORT` desfaz a transação corrente, fazendo com que todas as modificações realizadas por esta transação sejam desconsideradas. Este comando possui comportamento idêntico ao do comando SQL padrão *ROLLBACK*, estando presente apenas por motivos históricos.

## Parâmetros

`WORK`

`TRANSACTION`

Palavras chave opcionais. Não produzem nenhum efeito.

## Observações

Use o comando *COMMIT* para terminar uma transação bem-sucedida.

A utilização do `ABORT` fora de uma transação não causa nenhum problema, mas causa uma mensagem de advertência.

## Exemplos

Para anular todas as modificações:

```
ABORT ;
```

## Compatibilidade

Este comando é uma extensão do PostgreSQL presente por motivos históricos. O *ROLLBACK* é o comando equivalente do padrão SQL.

## Consulte também

*BEGIN*, *COMMIT*, *ROLLBACK*

# ALTER AGGREGATE

## Nome

ALTER AGGREGATE — altera a definição de uma função de agregação

## Sinopse

```
ALTER AGGREGATE nome ( tipo ) RENAME TO novo_nome  
ALTER AGGREGATE nome ( tipo ) OWNER TO novo_dono
```

## Descrição

O comando ALTER AGGREGATE altera a definição de uma função de agregação.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) de uma função de agregação existente.

*tipo*

O tipo de dado do argumento da função de agregação, ou \* se a função aceitar qualquer tipo de dado.

*novo\_nome*

O novo nome da função de agregação.

*novo\_dono*

O novo dono da função de agregação. É necessário ser um superusuário para mudar o dono da função de agregação.

## Exemplos

Para mudar o nome da função de agregação minhamedia para o tipo integer para minha\_media:

```
ALTER AGGREGATE minhamedia(integer) RENAME TO minha_media;
```

Para mudar o dono da função de agregação minhamedia para o tipo integer para joel:

```
ALTER AGGREGATE minhamedia(integer) OWNER TO joel;
```

## Compatibilidade

Não existe o comando ALTER AGGREGATE no padrão SQL.

## Consulte também

CREATE AGGREGATE, DROP AGGREGATE

# ALTER CONVERSION

## Nome

ALTER CONVERSION — altera a definição de uma conversão de codificação

## Sinopse

```
ALTER CONVERSION nome RENAME TO novo_nome  
ALTER CONVERSION nome OWNER TO novo_dono
```

## Descrição

O comando ALTER CONVERSION altera a definição de uma conversão de codificação.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) de uma conversão existente.

*novo\_nome*

O novo nome da conversão.

*novo\_dono*

O novo dono da conversão. É necessário ser um superusuário para mudar o dono da conversão.

## Exemplos

Para mudar o nome da conversão de iso\_8859\_1\_to\_utf\_8 para latin1\_to\_unicode:

```
ALTER CONVERSION iso_8859_1_to_utf_8 RENAME TO latin1_to_unicode;
```

Mudar o dono da conversão iso\_8859\_1\_to\_utf\_8 para joel:

```
ALTER CONVERSION iso_8859_1_to_utf_8 OWNER TO joel;
```

## Compatibilidade

Não existe o comando ALTER CONVERSION no padrão SQL.

## Consulte também

CREATE CONVERSION, DROP CONVERSION



# ALTER DATABASE

## Nome

ALTER DATABASE — altera um banco de dados

## Sinopse

```
ALTER DATABASE nome SET parâmetro { TO | = } { valor | DEFAULT }  
ALTER DATABASE nome RESET parâmetro  
ALTER DATABASE nome RENAME TO novo_nome  
ALTER DATABASE nome OWNER TO novo_dono
```

## Descrição

O comando ALTER DATABASE altera os atributos de um banco de dados.

As duas primeiras formas mudam, para um banco de dados do PostgreSQL, o valor padrão para a sessão de uma variável de configuração em tempo de execução. Depois, sempre que uma nova sessão for iniciada neste banco de dados, o valor especificado se torna o valor padrão para a sessão. O padrão específico para o banco de dados substitui qualquer definição presente no arquivo `postgresql.conf`, ou que tenha sido recebida a partir da linha de comando do `postmaster`. Somente o dono do banco de dados ou um superusuário podem mudar os padrões para a sessão de um banco de dados. Certas variáveis não podem ser definidas desta maneira, ou somente podem ser definidas por um superusuário.

A terceira forma muda o nome do banco de dados. Somente o dono do banco de dados ou um superusuário podem mudar o nome do banco de dados; os donos que não são superusuários devem possuir, também, o privilégio `CREATEDB`. O banco de dados corrente não pode ter seu nome mudado (Deve-se conectar a um banco de dados diferente se for necessário realizar esta operação).

A quarta forma muda o dono do banco de dados. Somente um superusuário pode mudar o dono do banco de dados.

## Parâmetros

*nome*

O nome do banco de dados cujos atributos estão sendo alterados.

*parâmetro*

*valor*

Define o padrão de sessão deste banco de dados, para o parâmetro de configuração especificado, como o valor fornecido. Se *valor* for `DEFAULT` ou, de forma equivalente, se `RESET` for utilizado, a definição específica para o banco de dados é removida, e a definição padrão global do sistema passa a ser herdada nas novas sessões. Deve ser utilizado `RESET ALL` para remover todas as definições específicas do banco de dados.

Consulte o comando *SET* e a Seção 16.4 para obter informações adicionais sobre os nomes e valores permitidos para os parâmetros.

*novo\_nome*

O novo nome do banco de dados.

*novo\_dono*

O novo dono do banco de dados.

## Observações

Também é possível ligar o padrão de sessão a um usuário específico, em vez de a um banco de dados; consulte o comando *ALTER USER*. As definições específicas para o usuário substituem as definições específicas para o banco de dados, no caso de haver conflito.

## Exemplos

Para desabilitar a varredura de índices no banco de dados teste por padrão:

```
ALTER DATABASE teste SET enable_indexscan TO off;
```

## Compatibilidade

O comando ALTER DATABASE é uma extensão do PostgreSQL.

## Consulte também

*CREATE DATABASE, DROP DATABASE, SET*

# ALTER DOMAIN

## Nome

ALTER DOMAIN — altera a definição de um domínio

## Sinopse

```
ALTER DOMAIN nome
    { SET DEFAULT expressão | DROP DEFAULT }
ALTER DOMAIN nome
    { SET | DROP } NOT NULL
ALTER DOMAIN nome
    ADD restrição_de_domínio
ALTER DOMAIN nome
    DROP CONSTRAINT nome_da_restrição [ RESTRICT | CASCADE ]
ALTER DOMAIN nome
    OWNER TO novo_dono
```

## Descrição

O comando ALTER DOMAIN altera a definição de um domínio existente. Existem várias subformas:

### SET/DROP DEFAULT

Estas formas definem ou removem o valor padrão do domínio. Deve ser observado que estes padrões somente se aplicam aos comandos INSERT subsequentes; não são afetadas as linhas já existentes nas tabelas que utilizam o domínio.

### SET/DROP NOT NULL

Estas formas modificam se o domínio está marcado para permitir valores nulos ou para rejeitar valores nulos. O comando SET NOT NULL pode ser utilizado somente quando as colunas que utilizam o domínio não contêm valores nulos.

### ADD *restrição\_de\_domínio*

Esta forma adiciona uma nova restrição ao domínio, utilizando a mesma sintaxe de CREATE DOMAIN. Somente será bem-sucedida se todas as colunas que utilizam o domínio satisfizerem a nova restrição.

### DROP CONSTRAINT

Esta forma remove restrições no domínio.

### OWNER

Esta forma torna o usuário especificado o dono do domínio.

É necessário ser o dono do domínio para utilizar ALTER DOMAIN; exceto no caso de ALTER DOMAIN OWNER, que só pode ser executado por um superusuário.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) do domínio existente a ser alterado.

*restrição\_de\_domínio*

Nova restrição de domínio para o domínio.

*nome\_da\_restrição*

Nome da restrição existente a ser removida.

CASCADE

Remove, automaticamente, os objetos que dependem da restrição.

RESTRICT

Não permite remover a restrição caso existam objetos que dependam da mesma. Este é o comportamento padrão.

*novo\_dono*

O nome de usuário do novo dono do domínio.

## Exemplos

Para adicionar a restrição NOT NULL ao domínio:

```
ALTER DOMAIN cep SET NOT NULL;
```

Para remover a restrição NOT NULL do domínio:

```
ALTER DOMAIN cep DROP NOT NULL;
```

Para adicionar uma restrição de verificação ao domínio:

```
ALTER DOMAIN cep ADD CONSTRAINT chk_cep CHECK (char_length(VALUE) = 8);
```

Para remover uma restrição de verificação do domínio:

```
ALTER DOMAIN cep DROP CONSTRAINT chk_cep;
```

## Compatibilidade

O comando ALTER DOMAIN é compatível com o SQL:1999, exceto pela variante OWNER, que é uma extensão do PostgreSQL.

# ALTER FUNCTION

## Nome

ALTER FUNCTION — altera a definição de uma função

## Sinopse

```
ALTER FUNCTION nome ( [ tipo [, ...] ] ) RENAME TO novo_nome  
ALTER FUNCTION nome ( [ tipo [, ...] ] ) OWNER TO novo_dono
```

## Descrição

O comando ALTER FUNCTION altera a definição de uma função.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) de uma função existente.

*tipo*

O tipo de dado do argumento da função.

*novo\_nome*

O novo nome da função.

*novo\_dono*

O novo dono da função. Para mudar o dono da função é necessário ser um superusuário. Deve ser observado que, se a função estiver marcada com SECURITY DEFINER, subsequenteiramente esta será executada com o novo dono.

## Exemplos

Para mudar o nome da função sqrt para o tipo integer para raiz\_quadrada:

```
ALTER FUNCTION sqrt(integer) RENAME TO raiz_quadrada;
```

Para mudar o dono da função sqrt para o tipo integer para joel:

```
ALTER FUNCTION sqrt(integer) OWNER TO joel;
```

## Compatibilidade

Existe o comando ALTER FUNCTION no padrão SQL, mas não possui a opção de mudar o nome ou o dono da função.

## Consulte também

CREATE FUNCTION, DROP FUNCTION

# ALTER GROUP

## Nome

ALTER GROUP — altera um grupo de usuários

## Sinopse

```
ALTER GROUP nome_do_grupo ADD USER nome_do_usuario [, ... ]
ALTER GROUP nome_do_grupo DROP USER nome_do_usuario [, ... ]
ALTER GROUP nome_do_grupo RENAME TO novo_nome
```

## Descrição

O comando ALTER GROUP altera os atributos de um grupo de usuários.

As duas primeiras variantes adicionam ou removem usuários de um grupo, respectivamente. Somente os superusuários do banco de dados podem utilizar este comando.

A terceira forma muda o nome do grupo. Somente os superusuários do banco de dados podem mudar o nome de um grupo.

## Parâmetros

*nome\_do\_grupo*

O nome do grupo a ser modificado.

*nome\_do\_usuario*

Os usuários a serem adicionados ou removidos do grupo. Os usuários devem existir; o comando ALTER GROUP não cria sem remove usuários.

*novo\_nome*

O novo nome do grupo.

## Exemplos

Adicionar usuários a um grupo:

```
ALTER GROUP arquitetura ADD USER joana, alberto;
```

Remover um usuário de um grupo:

```
ALTER GROUP engenharia DROP USER margarida;
```

## Compatibilidade

Não existe o comando ALTER GROUP no padrão SQL. O conceito de papéis (“roles”) é semelhante ao de grupos.

## Consulte também

CREATE GROUP, DROP GROUP

# ALTER INDEX

## Nome

ALTER INDEX — altera a definição de um índice

## Sinopse

```
ALTER INDEX nome
    ação [, ... ]
ALTER INDEX nome
    RENAME TO novo_nome
```

onde *ação* é um entre:

```
OWNER TO novo_dono
SET TABLESPACE nome_do_espaco_de_indices
```

## Descrição

O comando ALTER INDEX altera a definição de um índice existente. Existem diversas subformas:

OWNER

Esta forma torna o usuário especificado o dono do índice. Somente pode ser utilizado por um superusuário.

SET TABLESPACE

Esta forma altera o espaço de tabelas do índice para o espaço de tabelas especificado, e move os arquivos de dados associados ao índice para o novo espaço de tabelas. Consulte também *CREATE TABLESPACE*.

RENAME

A forma RENAME muda o nome do índice. Não há efeito sobre os dados armazenados.

Todas as ações, com exceção de RENAME, podem ser combinadas em uma lista de alterações múltiplas a serem aplicadas em paralelo.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) de um índice existente.

*novo\_nome*

O novo nome do índice.

*novo\_dono*

O nome de usuário do novo dono do índice.

*nome\_do\_espaco\_de\_tabelas*

O nome do espaço de tabelas para o qual o índice será movido.

## Observações

Estas operações também podem ser feitas utilizando *ALTER TABLE*. O comando ALTER INDEX é, na verdade, apenas um sinônimo para as formas de ALTER TABLE que se aplicam aos índices.

Não é permitido alterar qualquer parte dos catálogos do sistema.

## Exemplos

Para mudar o nome de um índice existente:

```
ALTER INDEX distribuidores RENAME TO fornecedores;
```

Para mover um índice para outro espaço de tabelas:

```
ALTER INDEX distribuidores SET TABLESPACE espaco_de_tabelas_rapido;
```

## Compatibilidade

O comando `ALTER INDEX` é uma extensão do PostgreSQL.



# ALTER LANGUAGE

## Nome

ALTER LANGUAGE — altera a definição de uma linguagem procedural

## Sinopse

```
ALTER LANGUAGE nome RENAME TO novo_nome
```

## Descrição

O comando ALTER LANGUAGE altera a definição de uma linguagem. A única funcionalidade é mudar o nome da linguagem. Somente um superusuário pode mudar o nome de uma linguagem.

## Parâmetros

*nome*

O nome da linguagem

*novo\_nome*

O novo nome da linguagem

## Compatibilidade

Não existe o comando ALTER LANGUAGE no padrão SQL.

## Consulte também

CREATE LANGUAGE, DROP LANGUAGE

# ALTER OPERATOR

## Nome

ALTER OPERATOR — altera a definição de um operador

## Sinopse

```
ALTER OPERATOR nome ( { tipo_à_esquerda | NONE } , { tipo_à_direita | NONE } ) OWNER TO novo_dono
```

## Descrição

O comando ALTER OPERATOR altera a definição de um operador. Atualmente a única funcionalidade disponível é mudar o dono do operador.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) de um operador existente.

*tipo\_à\_esquerda*

O tipo de dado do operando à esquerda do operador; deve-se escrever NONE se o operador não tiver operando à esquerda.

*tipo\_à\_direita*

O tipo de dado do operando à direita do operador; deve-se escrever NONE se o operador não tiver operando à direita.

*novo\_dono*

O novo dono da operador. Para mudar o dono do operador é necessário ser um superusuário.

## Exemplos

Mudar o dono do operador personalizado a @@ b para o tipo text:

```
ALTER OPERATOR @@ (text, text) OWNER TO joel;
```

## Compatibilidade

Não existe o comando ALTER OPERATOR no padrão SQL.

## Consulte também

CREATE OPERATOR, DROP OPERATOR

# ALTER OPERATOR CLASS

## Nome

ALTER OPERATOR CLASS — altera a definição de uma classe de operadores

## Sinopse

```
ALTER OPERATOR CLASS nome USING método_de_índice RENAME TO novo_nome  
ALTER OPERATOR CLASS nome USING método_de_índice OWNER TO novo_dono
```

## Descrição

O comando ALTER OPERATOR CLASS altera a definição de uma classe de operadores.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) de uma classe de operadores existente.

*método\_de\_índice*

O nome do método de índice para o qual esta classe de operadores se destina.

*novo\_nome*

O novo nome da classe de operadores.

*novo\_dono*

O nome de usuário do novo dono do índice. É necessário ser um superusuário para mudar o dono de uma classe de operadores.

## Compatibilidade

Não existe o comando ALTER OPERATOR CLASS no padrão SQL.

## Consulte também

CREATE OPERATOR CLASS, DROP OPERATOR CLASS

# ALTER SCHEMA

## Nome

ALTER SCHEMA — altera a definição de um esquema

## Sinopse

ALTER SCHEMA *nome* RENAME TO *novo\_nome*

ALTER SCHEMA *nome* OWNER TO *novo\_dono*

## Descrição

O comando ALTER SCHEMA altera a definição de um esquema. Para mudar o nome do esquema é necessário ser o dono do esquema e possuir o privilégio CREATE para o banco de dados. Para mudar o dono do esquema é necessário ser um superusuário.

## Parâmetros

*nome*

O nome de um esquema existente.

*novo\_nome*

O novo nome do esquema. O novo nome não pode começar por pg\_, porque estes nomes são reservados para os esquemas do sistema.

*novo\_dono*

O novo dono do esquema.

## Compatibilidade

Não existe o comando ALTER SCHEMA no padrão SQL.

## Consulte também

CREATE SCHEMA, DROP SCHEMA

# ALTER SEQUENCE

## Nome

ALTER SEQUENCE — altera a definição de um gerador de seqüência

## Sinopse

```
ALTER SEQUENCE nome [ INCREMENT [ BY ] incremento ]  
    [ MINVALUE valor_mínimo | NO MINVALUE ] [ MAXVALUE valor_máximo | NO MAXVALUE ]  
    [ RESTART [ WITH ] início ] [ CACHE cache ] [ [ NO ] CYCLE ]
```

## Descrição

O comando ALTER SEQUENCE altera os parâmetros de um gerador de seqüência existente. Todos os parâmetros que não são explicitamente definidos no comando ALTER SEQUENCE mantêm suas definições anteriores.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) da seqüência a ser alterada.

*incremento*

A cláusula INCREMENT BY *incremento* é opcional. Um valor positivo produz uma seqüência ascendente, enquanto um valor negativo produz uma seqüência descendente. Se não for especificado, o valor anterior do incremento será mantido.

*valor\_mínimo*

NO MINVALUE

A cláusula opcional MINVALUE *valor\_mínimo* especifica o valor mínimo para a seqüência. Se for especificado NO MINVALUE, serão utilizados os valores padrão 1 e  $-2^{63}-1$  para seqüências ascendentes e descendentes, respectivamente. Se nenhuma das duas opções for especificada, o valor mínimo corrente será mantido.

*valor\_máximo*

NO MAXVALUE

A cláusula opcional MAXVALUE *valor\_máximo* especifica o valor máximo para a seqüência. Se for especificado NO MAXVALUE, serão utilizados os valores padrão  $2^{63}-1$  e -1, para seqüências ascendentes e descendentes, respectivamente. Se nenhuma das duas opções for especificada, o valor máximo corrente será mantido.

*início*

A cláusula opcional RESTART WITH *início* muda o valor corrente da seqüência.

*cache*

A cláusula CACHE *cache* permite que números da seqüência sejam pré-alocados e armazenados em memória para obter um acesso mais rápido. O valor mínimo é 1 (somente um valor pode ser gerado de cada vez, ou seja, sem cache). Se não for especificado, o valor anterior de cache será mantido.

CYCLE

A palavra chave opcional CYCLE pode ser utilizada para permitir uma seqüência ascendente ou descendente reiniciar quando atingir o *valor\_máximo* ou o *valor\_mínimo*, respectivamente. Se o limite for atingido, o próximo número gerado será o *valor\_mínimo* ou o *valor\_máximo*, respectivamente.

NO CYCLE

Se a palavra chave opcional NO CYCLE for especificada, todas as chamadas a nextval após a seqüência ter atingido seu valor máximo retornam um erro. Se nem CYCLE nem NO CYCLE for especificado, o comportamento anterior para o ciclo será mantido.

## Exemplos

Reiniciar uma sequência chamada `serial`, em 105:

```
ALTER SEQUENCE serial RESTART WITH 105;
```

## Observações

Para evitar o bloqueio de transações concorrentes que obtêm números a partir de uma mesma sequência, o comando `ALTER SEQUENCE` nunca é desfeito (`rollback`); as mudanças entram em vigor imediatamente, não sendo reversíveis.

O comando `ALTER SEQUENCE` não afeta imediatamente os resultados de `nextval` nos processos servidor que pré-alocaram (`cached`) valores da sequência, a não ser no processo servidor corrente. Os demais processos servidor utilizarão todos os valores pré-alocados antes de observar a mudança dos parâmetros da sequência. O processo servidor corrente é afetado imediatamente.

## Compatibilidade

O comando `ALTER SEQUENCE` está em conformidade com o SQL:2003.

# ALTER TABLE

## Nome

ALTER TABLE — altera a definição de uma tabela

## Sinopse

```
ALTER TABLE [ ONLY ] nome [ * ]
    ação [, ... ]
ALTER TABLE [ ONLY ] nome [ * ]
    RENAME [ COLUMN ] coluna TO novo_nome_da_coluna
ALTER TABLE nome
    RENAME TO novo_nome
```

onde *ação* é uma entre:

```
ADD [ COLUMN ] coluna tipo [ restrição_de_coluna [ ... ] ]
DROP [ COLUMN ] coluna [ RESTRICT | CASCADE ]
ALTER [ COLUMN ] coluna TYPE tipo [ USING expressão ]
ALTER [ COLUMN ] coluna SET DEFAULT expressão
ALTER [ COLUMN ] coluna DROP DEFAULT
ALTER [ COLUMN ] coluna { SET | DROP } NOT NULL
ALTER [ COLUMN ] coluna SET STATISTICS inteiro
ALTER [ COLUMN ] coluna SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
ADD restrição_de_tabela
DROP CONSTRAINT nome_da_restrição [ RESTRICT | CASCADE ]
CLUSTER ON nome_do_índice
SET WITHOUT CLUSTER
SET WITHOUT OIDS
OWNER TO novo_dono
SET TABLESPACE nome_do_espaco_de_tabelas
```

## Descrição

O comando ALTER TABLE altera a definição de uma tabela existente. Existem várias sub-formas:

ADD COLUMN

Esta forma adiciona uma nova coluna à tabela utilizando a mesma sintaxe do comando *CREATE TABLE*.

DROP COLUMN

Esta forma remove uma coluna da tabela. Os índices e as restrições da tabela que envolvem a coluna também são automaticamente removidos. É necessário especificar *CASCADE* se algum objeto fora da tabela depender da coluna como, por exemplo, referências de chaves estrangeiras ou visões.

ALTER COLUMN TYPE

Esta forma muda o tipo de uma coluna da tabela. Os índices e as restrições de tabela simples que envolvem a coluna são automaticamente convertidos para usar o novo tipo da coluna, através da reanálise da expressão original fornecida. A cláusula opcional *USING* especifica como computar o novo valor da coluna a partir do antigo; quando omitida, a conversão padrão é a mesma de uma conversão de atribuição do tipo antigo para o novo. A cláusula *USING* deve ser fornecida quando não há nenhuma conversão implícita ou de atribuição do tipo antigo para o novo.

SET/DROP DEFAULT

Estas formas definem ou removem o valor padrão para a coluna. O valor padrão somente é aplicado aos comandos *INSERT* subsequentes; não modifica as linhas existentes na tabela. Valores padrão também podem ser criados para visões e, neste caso, são inseridos dentro do comando *INSERT* na visão, antes da regra *ON INSERT* da visão ser aplicada.

## SET/DROP NOT NULL

Estas formas alteram se a coluna está marcada para permitir valores nulos ou para rejeitar valores nulos. A forma `SET NOT NULL` só pode ser utilizada quando não existem valores nulos na coluna.

## SET STATISTICS

Esta forma define a quantidade de informações armazenadas na coleta de estatísticas por coluna para as operações subsequentes de *ANALYZE*. A quantidade pode ser definida no intervalo de 0 a 1000; como alternativa, pode ser definida como -1 para utilizar a quantidade de estatísticas padrão do sistema (`default_statistics_target`). Para obter informações adicionais sobre a utilização de estatísticas pelo planejador de comandos do PostgreSQL consulte a Seção 13.2.

## SET STORAGE

Esta forma define o modo de armazenamento da coluna. Controla se a coluna é mantida na mesma tabela ou em uma tabela suplementar, e se os dados devem ser comprimidos ou não. Deve ser utilizado `PLAIN` para valores de comprimento fixo, como `integer`, e fica na mesma tabela não comprimido. `MAIN` é utilizado para dados que ficam na mesma tabela e são compressíveis. `EXTERNAL` é utilizado para dados externos não comprimidos, e `EXTENDED` é utilizado para dados externos comprimidos. `EXTENDED` é o padrão para a maioria dos tipos de dado que suportam armazenamento não-`PLAIN`. A utilização de `EXTERNAL` torna as operações de substring em colunas `text` e `bytea` mais rápidas, às custas de um aumento no espaço para armazenamento. Deve ser observado que `SET STORAGE`, por si só, não muda nada na tabela, apenas define a estratégia a ser seguida durante as próximas atualizações da tabela. Consulte a Seção 49.2 para obter informações adicionais.

## ADD restrição\_de\_tabela

Esta forma adiciona uma nova restrição à tabela utilizando a mesma sintaxe do comando *CREATE TABLE*.

## DROP CONSTRAINT

Esta forma remove restrições de tabela. Atualmente as restrições de tabela não necessitam ter nomes únicos e, portanto, pode haver mais de uma restrição correspondendo ao nome especificado. Todas as restrições correspondentes são removidas.

## CLUSTER

Esta forma seleciona o índice padrão para as próximas operações de *CLUSTER*. Não efetua realmente o reagrupamento da tabela.

## SET WITHOUT CLUSTER

Esta forma remove da tabela a especificação do índice usado mais recentemente em *CLUSTER*. Afeta as próximas operações de agrupamento que não especificarem um índice.

## SET WITHOUT OIDS

Esta forma remove da tabela a coluna de sistema `oid`. É exatamente equivalente a `DROP COLUMN oid RESTRICT`, exceto que não reclama se já não houver mais a coluna `oid`.

Deve ser observado que não existe uma forma alternativa de `ALTER TABLE` que permita restaurar os OIDs para a tabela após estes terem sido removidos.

## OWNER

Esta forma torna o usuário especificado o dono da tabela, índice, sequência ou visão.

## SET TABLESPACE

Esta forma muda o espaço de tabelas da tabela para o espaço de tabelas especificado, e move os arquivos de dados associados à tabela para o novo espaço de tabelas. Havendo índices na tabela, estes não são movidos; porém, podem ser movidos separadamente através de comandos `SET TABLESPACE` adicionais. Consulte também *CREATE TABLESPACE*.

## RENAME

A forma `RENAME` muda o nome de uma tabela (de um índice, de uma sequência ou de uma visão), ou o nome de uma coluna da tabela. Não produz efeito sobre os dados armazenados.



Todas as ações, exceto RENAME, podem ser combinadas em uma lista de alterações múltiplas a serem aplicadas em paralelo. Por exemplo, é possível adicionar várias colunas e/ou alterar o tipo de várias colunas em um único comando. Esta situação é particularmente útil em tabelas grandes, uma vez que somente é necessário realizar uma passagem pela tabela.

É necessário ser o dono da tabela para executar ALTER TABLE; exceto para ALTER TABLE OWNER, que somente pode ser executado por um superusuário.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) da tabela existente a ser alterada. Se ONLY for especificado, somente esta tabela será alterada. Se ONLY não for especificado, a tabela e todas as suas tabelas descendentes (se existirem) são alteradas. O \* pode ser adicionado ao nome da tabela para indicar que as tabelas descendentes devem ser alteradas, mas na versão atual este é comportamento padrão (Nas versões anteriores a 7.1 ONLY era o comportamento padrão. O padrão pode ser alterado mudando o parâmetro de configuração sql\_inheritance.)

*coluna*

O nome de uma coluna nova ou existente.

*novo\_nome\_da\_coluna*

O novo nome para uma coluna existente.

*novo\_nome*

O novo nome da tabela.

*tipo*

O tipo de dado da nova coluna, ou o novo tipo de dado de uma coluna existente.

*restrição\_de\_tabela*

A nova restrição de tabela para a tabela.

*nome\_da\_restrição*

O nome da restrição existente a ser removida.

CASCADE

Remove, automaticamente, os objetos que dependem da coluna ou da restrição removida (por exemplo, visões fazendo referência à coluna).

RESTRICT

Não permite remover a coluna ou a restrição caso existam objetos que dependam das mesmas. Este é o comportamento padrão.

*nome\_do\_índice*

O nome do índice pelo qual a tabela deve ser marcada para agrupamento.

*novo\_dono*

O nome de usuário do novo dono da tabela.

*nome\_do\_espaco\_de\_tabelas*

O nome do espaço de tabelas para o qual a tabela será movida.

## Observações

A palavra chave COLUMN é apenas informativa, podendo ser omitida.

Quando uma coluna é adicionada usando ADD COLUMN, todas as linhas existentes na tabela são inicializadas com o valor padrão da coluna (NULL se a cláusula DEFAULT não for especificada).

Adicionar uma coluna com um valor padrão não nulo, ou mudar o tipo de uma coluna existente, faz com que toda a tabela seja reescrita. Isto pode levar uma quantidade de tempo significativa no caso de uma tabela grande; temporariamente será necessário o dobro do espaço em disco.

A adição das restrições CHECK e NOT NULL obrigam varrer toda a tabela para verificar se as linhas existentes estão em conformidade com a restrição.

A razão principal para fornecer a opção de especificar várias alterações em um único comando ALTER TABLE, é que várias varreduras ou reescritas da tabela podem assim ser combinadas em uma única passagem pela tabela.

A forma DROP COLUMN não remove fisicamente a coluna, simplesmente torna a coluna invisível para as operações SQL. As operações subsequentes de inclusão e de atualização na tabela armazenam o valor nulo na coluna. Portanto, remover uma coluna é rápido mas não reduz imediatamente o espaço em disco da tabela, porque o espaço ocupado pela coluna removida não é recuperado. O espaço é recuperado ao longo do tempo, à medida que as linhas existentes são atualizadas.

Algumas vezes é vantajoso o fato de ALTER TYPE requerer a reescrita de toda a tabela, porque o processo de reescrita elimina todo espaço morto presente na tabela. Por exemplo, para recuperar o espaço ocupado por uma coluna removida, a forma mais rápida é

```
ALTER TABLE tabela ALTER COLUMN qualquer_coluna TYPE qualquer_tipo;
```

onde qualquer\_coluna é uma coluna remanescente na tabela, e qualquer\_tipo é o mesmo tipo que a coluna já possui. Resulta em uma modificação da tabela que não é semanticamente visível, mas o comando força a reescrita que elimina os dados que não são mais úteis.

A opção USING de ALTER TYPE pode, na verdade, especificar qualquer expressão envolvendo os valores antigos da linha; ou seja, pode fazer referência a outras colunas assim como à coluna sendo convertida. Isto permite fazer conversões muito gerais através da sintaxe de ALTER TYPE. Por causa desta flexibilidade, a expressão do USING não é aplicada ao valor padrão da coluna (se houver); o resultado pode não ser uma expressão constante conforme requerido por um valor padrão. Isto significa que quando não existe conversão implícita ou de atribuição do tipo antigo para o novo, ALTER TYPE pode falhar na conversão do valor padrão, mesmo que a cláusula USING seja fornecida. Neste caso, deve ser removido o valor padrão utilizando DROP DEFAULT, executar ALTER TYPE e, então, usar SET DEFAULT para adicionar um novo valor padrão adequado. Considerações semelhantes se aplicam a índices e restrições envolvendo a coluna.

Se a tabela possuir tabelas descendentes não é permitido adicionar, mudar o nome ou mudar o tipo de uma coluna na tabela ancestral sem fazer o mesmo nas tabelas descendentes. Ou seja, ALTER TABLE ONLY será rejeitado. Isto garante que as tabelas descendentes sempre possuem colunas correspondendo às tabelas ancestrais.

Uma operação DROP COLUMN recursiva remove a coluna da tabela descendente somente se a tabela descendente não herdar esta coluna de outra tabela ancestral, e nunca tiver possuído uma definição independente da coluna. O DROP COLUMN não recursivo (ou seja, ALTER TABLE ONLY ... DROP COLUMN) nunca remove qualquer coluna de tabela descendente; em vez disso marca a coluna como definida de forma independente em vez de herdada.

Não é permitido alterar qualquer parte dos catálogos do sistema.

Consulte o comando CREATE TABLE para obter informações adicionais sobre os parâmetros válidos. O Capítulo 5 possui informações adicionais sobre herança.

## Exemplos

Para adicionar uma coluna do tipo varchar a uma tabela:

```
ALTER TABLE distribuidores ADD COLUMN endereco varchar(30);
```

Para excluir uma coluna da tabela:

```
ALTER TABLE distribuidores DROP COLUMN endereco RESTRICT;
```

Para mudar o tipo de duas colunas existentes em uma única operação:

```
ALTER TABLE distribuidores
    ALTER COLUMN endereco TYPE varchar(80),
    ALTER COLUMN nome TYPE varchar(100);
```

Para mudar uma coluna inteira do UNIX contendo carimbo do tempo para timestamp with time zone através da cláusula USING:

```
ALTER TABLE foo
  ALTER COLUMN foo_timestamp TYPE timestamp with time zone
  USING
    timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';
```

Para mudar o nome de uma coluna existente:

```
ALTER TABLE distribuidores RENAME COLUMN endereco TO cidade;
```

Para mudar o nome de uma tabela existente:

```
ALTER TABLE distribuidores RENAME TO fornecedores;
```

Para adicionar uma restrição de não nulo a uma coluna:

```
ALTER TABLE distribuidores ALTER COLUMN logradouro SET NOT NULL;
```

Para remover a restrição de não nulo da coluna:

```
ALTER TABLE distribuidores ALTER COLUMN logradouro DROP NOT NULL;
```

Para adicionar uma restrição de verificação à tabela:

```
ALTER TABLE distribuidores ADD CONSTRAINT chk_cep CHECK (char_length(cod_cep) = 8);
```

Para remover uma restrição de verificação de uma tabela e de todas as suas descendentes:

```
ALTER TABLE distribuidores DROP CONSTRAINT chk_cep;
```

Para adicionar uma restrição de chave estrangeira a uma tabela:

```
ALTER TABLE distribuidores ADD CONSTRAINT fk_dist FOREIGN KEY (endereco) REFERENCES
enderecos (endereco) MATCH FULL;
```

Para adicionar uma restrição de unicidade (multicoluna) à tabela:

```
ALTER TABLE distribuidores ADD CONSTRAINT unq_id_dist_cod_cep UNIQUE (id_dist, cod_cep);
```

Para adicionar uma restrição de chave primária a uma tabela com o nome gerado automaticamente, levando em conta que a tabela pode possuir somente uma única chave primária:

```
ALTER TABLE distribuidores ADD PRIMARY KEY (id_dist);
```

Para mover a tabela para outro espaço de tabelas:

```
ALTER TABLE distribuidores SET TABLESPACE espaco_de_tabelas_rapido;
```

## Compatibilidade

As formas ADD, DROP e SET DEFAULT estão em conformidade com o padrão SQL. As outras formas são extensões do PostgreSQL ao padrão SQL. Também, a capacidade de especificar mais de uma manipulação em um único comando ALTER TABLE é uma extensão.

O comando ALTER TABLE DROP COLUMN pode ser utilizado para remover a única coluna da tabela, produzindo uma tabela com zero coluna. Esta é uma extensão ao SQL, que não permite tabelas sem nenhuma coluna.

# ALTER TABLESPACE

## Nome

ALTER TABLESPACE — altera a definição de um espaço de tabelas

## Sinopse

```
ALTER TABLESPACE nome RENAME TO novo_nome  
ALTER TABLESPACE nome OWNER TO novo_dono
```

## Descrição

O comando ALTER TABLESPACE altera a definição de um espaço de tabelas.

## Parâmetros

*nome*

O nome de um espaço de tabelas existente.

*novo\_nome*

O novo nome do espaço de tabelas. O novo nome não pode começar por `pg_`, porque estes nomes são reservados para os espaços de tabela do sistema.

*novo\_dono*

O novo dono do espaço de tabelas. É necessário ser um superusuário para mudar o dono do espaço de tabelas.

## Exemplos

Mudar o nome de `espaco_para_indices` para `raid_rapido`:

```
ALTER TABLESPACE espaco_para_indices RENAME TO raid_rapido;
```

Mudar o dono do espaço de tabelas `espaco_para_indices`:

```
ALTER TABLESPACE espaco_para_indices OWNER TO maria;
```

## Compatibilidade

Não existe o comando ALTER TABLESPACE no padrão SQL.

## Consulte também

*CREATE TABLESPACE*, *DROP TABLESPACE*

# ALTER TRIGGER

## Nome

ALTER TRIGGER — altera a definição de um gatilho

## Sinopse

```
ALTER TRIGGER nome ON tabela RENAME TO novo_nome
```

## Descrição

O comando ALTER TRIGGER altera as propriedades de um gatilho existente. A cláusula RENAME muda o nome de um determinado gatilho sem mudar a definição do gatilho.

É necessário ser o dono da tabela onde o gatilho atua para poder mudar suas propriedades.

## Parâmetros

*nome*

O nome do gatilho existente a ser alterado.

*tabela*

O nome da tabela onde o gatilho atua.

*novo\_nome*

O novo nome do gatilho.

## Exemplo

Para mudar o nome de um gatilho existente:

```
ALTER TRIGGER emp_alt ON tbl_employees RENAME TO trg_altera_empregado;
```

## Compatibilidade

O comando ALTER TRIGGER é uma extensão do PostgreSQL ao padrão SQL.

# ALTER TYPE

## Nome

`ALTER TYPE` — altera a definição de um tipo

## Sinopse

```
ALTER TYPE nome OWNER TO novo_dono
```

## Descrição

O comando `ALTER TYPE` altera a definição de um tipo existente. Atualmente a única funcionalidade disponível é mudar o nome do tipo.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) do tipo existente a ser alterado.

*novo\_dono*

O nome de usuário do novo dono do tipo. É necessário ser um superusuário para mudar o dono do tipo.

## Exemplos

Para mudar o dono do tipo `email`, definido pelo usuário, para `joel`:

```
ALTER TYPE email OWNER TO joel;
```

## Compatibilidade

Não existe o comando `ALTER TYPE` no padrão SQL.

# ALTER USER

## Nome

ALTER USER — altera uma conta de usuário do banco de dados

## Sinopse

```
ALTER USER nome [ [ WITH ] opção [ ... ] ]
```

onde *opção* pode ser:

```
CREATEDB | NOCREATEDB
| CREATEUSER | NOCREATEUSER
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'senha'
| VALID UNTIL 'data_e_hora'
```

```
ALTER USER nome RENAME TO novo_nome
ALTER USER nome SET parâmetro { TO | = } { valor | DEFAULT }
ALTER USER nome RESET parâmetro
```

## Descrição

O comando ALTER USER altera os atributos de uma conta de usuário do PostgreSQL. Os atributos não mencionados no comando permanecem com suas definições anteriores.

A primeira variante deste comando listada na sinopse muda determinados privilégios e configurações de autenticação para um usuário (Veja abaixo para obter detalhes). Somente um superusuário do banco de dados pode alterar qualquer uma destas configurações para qualquer usuário. Os usuários comuns podem apenas alterar suas próprias senhas.

A segunda variante muda o nome do usuário. Somente um superusuário do banco de dados pode mudar nome de conta de usuário. O usuário da sessão corrente não pode ter o nome mudado (conecte como um usuário diferente se precisar fazer isto). Como as senhas criptografadas com MD5 utilizam o nome de usuário como sal criptográfico (`cryptographic salt`)<sup>1</sup> mudar o nome de usuário limpa sua senha MD5.

A terceira e a quarta variantes mudam, para uma determinada variável de configuração, o valor padrão da sessão do usuário. Após isto, sempre que o usuário iniciar uma nova sessão o valor especificado se tornará o padrão da sessão, substituindo qualquer configuração presente em `postgresql.conf`, ou que tenha sido recebida através da linha de comando do `postmaster`. Os usuários comuns podem mudar seus próprios padrões de sessão. Os superusuários podem mudar os padrões de sessão de qualquer usuário. Certas variáveis não podem ser definidas desta maneira, ou só podem ser definidas por um superusuário.

## Parâmetros

*nome*

O nome de usuário cujos atributos estão sendo alterados.

CREATEDB  
NOCREATEDB

Estas cláusulas definem a permissão para o usuário criar banco de dados. Se CREATEDB for especificado, o usuário terá permissão para criar seus próprios bancos de dados. Especificando NOCREATEDB nega-se ao usuário a permissão para criar banco de dados (se o usuário for um superusuário então esta definição não tem efeito prático).

CREATEUSER  
NOCREATEUSER

Estas cláusulas determinam se o usuário terá permissão para criar novos usuários. CREATEUSER torna o usuário um superusuário, não sujeito a restrições de acesso.

*senha*

A nova senha a ser utilizada para esta conta.

ENCRYPTED

UNENCRYPTED

Estas palavras chave controlam se a senha é armazenada criptografada, ou não, em `pg_shadow` (Consulte o comando *CREATE USER* para obter informações adicionais sobre esta opção).

*data\_e\_hora*

A data (e, opcionalmente, a hora) de expiração da senha do usuário. Para fazer com que a senha nunca expire deve ser utilizado `'infinity'`.

*novو\_nome*

O novo nome de usuário.

*parâmetro*

*valor*

Define o valor fornecido como sendo o valor padrão para o parâmetro de configuração especificado. Se *valor* for `DEFAULT` ou, de forma equivalente, se `RESET` for utilizado, a definição da variável específica para o usuário é removida, e o valor padrão global do sistema será herdado nas novas sessões do usuário. Use `RESET ALL` para remover todas as definições específicas do usuário.

Consulte o comando *SET* e a Seção 16.4 para obter informações adicionais sobre os nomes e valores permitidos para os parâmetros.

## Observações

Use o comando *CREATE USER* para criar novos usuários, e o comando *DROP USER* para remover um usuário.

O comando *ALTER USER* não pode mudar a participação do usuário nos grupos. Use o comando *ALTER GROUP* para realizar esta operação.

A cláusula `VALID UNTIL` define uma data de expiração para a senha apenas, e não para a conta do usuário *per se*. Em particular, a obediência à data de expiração não é imposta ao se conectar utilizando um método de autenticação não baseado em senha.

Também é possível ligar o padrão de sessão a um banco de dados específico em vez de a um usuário; consulte o comando *ALTER DATABASE*. As definições específicas para o usuário substituem as definições específicas para o banco de dados, no caso de haver conflito.

## Exemplos

Mudar a senha do usuário:

```
ALTER USER marcos WITH PASSWORD 'hu8jmn3';
```

Mudar a data de expiração da senha do usuário:

```
ALTER USER manuel VALID UNTIL 'Jan 31 2030';
```

Mudar a data de expiração da senha, especificando que a senha expira ao meio dia de 4 de maio de 2005, usando uma zona horária uma hora adiante da UTC:

```
ALTER USER cristiane VALID UNTIL 'May 4 12:00:00 2005 +1';
```

Tornar o usuário válido para sempre:

```
ALTER USER andrea VALID UNTIL 'infinity';
```

Dar ao usuário permissão para criar outros usuários e novos bancos de dados:

```
ALTER USER luizete CREATEUSER CREATEDB;
```



## Compatibilidade

O comando `ALTER USER` é uma extensão do PostgreSQL. O padrão SQL deixa a definição dos usuários para a implementação.

## Consulte também

*CREATE USER, DROP USER, SET*

## Notas

1. Em criptografia sal consiste de bits aleatórios (tipicamente 12 ou mais) usados como uma das entradas para a função de derivação da chave. Salt (cryptography) ([http://pedia.newsfiler.co.uk/wikipedia/s/sa/salt\\_\\_cryptography\\_.html](http://pedia.newsfiler.co.uk/wikipedia/s/sa/salt__cryptography_.html)). (N. do T.)

# ANALYZE

## Nome

ANALYZE — coleta estatísticas sobre o banco de dados

## Sinopse

```
ANALYZE [ VERBOSE ] [ tabela [ (coluna [, ...] ) ] ]
```

## Descrição

O comando `ANALYZE` coleta estatísticas sobre o conteúdo das tabelas do banco de dados, e armazena os resultados na tabela do sistema `pg_statistic`. Posteriormente, o planejador de comandos utiliza estas estatísticas para ajudar a determinar o plano de execução mais eficiente para os comandos.

Sem nenhum parâmetro, o comando `ANALYZE` analisa todas as tabelas do banco de dados corrente. Com um parâmetro, o comando `ANALYZE` analisa somente esta tabela. É possível, também, fornecer uma lista de nomes de colunas e, neste caso, somente são coletadas estatísticas para estas colunas.

## Parâmetros

`VERBOSE`

Habilita a exibição das mensagens de progresso.

`tabela`

O nome (possivelmente qualificado pelo esquema) da tabela a ser analisada. O padrão é analisar todas as tabelas do banco de dados corrente.

`coluna`

O nome de uma determinada coluna a ser analisada. O padrão é analisar todas as colunas.

## Saídas

Quando `VERBOSE` é especificado, o comando `ANALYZE` emite mensagens de progresso indicando qual tabela está sendo processada no momento. Várias estatísticas sobre as tabelas também são mostradas.

## Observações

Aconselha-se executar o comando `ANALYZE` periodicamente, ou logo após realizar uma alteração importante no conteúdo de uma tabela. Estatísticas precisas auxiliam o planejador na escolha do plano de comando mais apropriado e, portanto, melhoram o tempo do processamento do comando. Uma estratégia habitual é executar `VACUUM` e `ANALYZE` uma vez por dia em hora de pouca utilização.

Ao contrário do comando `VACUUM FULL`, o comando `ANALYZE` requer somente um bloqueio de leitura na tabela podendo, portanto, ser executado em conjunto com outras atividades na tabela.

As estatísticas coletadas pelo comando `ANALYZE` geralmente incluem a lista dos valores mais comuns de cada coluna, e um histograma mostrando a distribuição aproximada dos dados em cada coluna. Estas informações podem ser omitidas se o comando `ANALYZE` considerá-las sem importância (por exemplo, em uma coluna de chave única não existem valores repetidos), ou se o tipo de dado da coluna não suportar os operadores apropriados. Existem informações adicionais sobre estatísticas no Capítulo 21.

Para tabelas grandes, o comando `ANALYZE` pega amostras aleatórias do conteúdo da tabela, em vez de examinar todas as linhas. Esta estratégia permite que mesmo tabelas muito grandes sejam analisadas em curto espaço de tempo. Entretanto, deve ser observado que as estatísticas são apenas aproximadas, e mudam um pouco cada vez que o comando `ANALYZE` é executado, mesmo que o conteúdo da tabela não se altere, podendo provocar pequenas mudanças no custo estimado pelo planejador mostrado pelo comando `EXPLAIN`. Em situações raras, este não determinismo faz o otimizador de consultas

escolher planos diferentes entre execuções do comando `ANALYZE`. Para evitar esta situação, a quantidade de estatísticas coletada pelo comando `ANALYZE` deve ser aumentada, conforme descrito abaixo.

A extensão da análise pode ser controlada ajustando o valor da variável de configuração `default_statistics_target`, ou coluna por coluna definindo a quantidade de estatísticas por coluna através do comando `ALTER TABLE ... ALTER COLUMN ... SET STATISTICS` (consulte o comando `ALTER TABLE`). A quantidade especificada define o número máximo de entradas presentes na lista de valores com maior incidência, e o número máximo de barras no histograma. O valor padrão é 10, mas pode ser ajustado para mais, ou para menos, para balancear a precisão das estimativas do planejador contra o tempo gasto para executar o comando `ANALYZE` e a quantidade de espaço ocupado pela tabela `pg_statistic`. Em particular, especificar o valor zero desabilita a coleta de estatísticas para a coluna, podendo ser útil em colunas que nunca são usadas como parte das cláusulas `WHERE`, `GROUP BY` ou `ORDER BY` nos comandos, porque as estatísticas para estas colunas nunca são utilizadas pelo planejador.

A quantidade máxima de estatísticas entre as colunas sendo analisadas determina o número de linhas amostradas para preparar as estatísticas. Aumentando a quantidade aumenta proporcionalmente o tempo e espaço necessários para executar o comando `ANALYZE`.

## Compatibilidade

Não existe o comando `ANALYZE` no padrão SQL.

# BEGIN

## Nome

BEGIN — inicia um bloco de transação

## Sinopse

```
BEGIN [ WORK | TRANSACTION ] [ modo_da_transação [, ...] ]
```

onde *modo\_da\_transação* é um entre:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }  
READ WRITE | READ ONLY
```

## Descrição

O comando `BEGIN` inicia um bloco de transação, ou seja, todos os comandos após o `BEGIN` são executados em uma única transação, até ser encontrado um `COMMIT` ou `ROLLBACK` explícito. Por padrão (sem o `BEGIN`), o PostgreSQL executa as transações no modo de auto-efetivação (`autocommit`), ou seja, cada comando é executado em sua própria transação e uma efetivação (`commit`) é realizada implicitamente no fim do comando, se a execução for bem-sucedida, senão a transação é desfeita (`rollback`).

Os comandos são executados mais rapidamente em um bloco de transação, porque cada início/efetivação de transação requer uma atividade significativa de CPU e de disco. A execução de vários comandos dentro de uma transação também é útil para garantir a consistência ao se fazer várias modificações relacionadas: as outras sessões não conseguem ver os estados intermediários até que todas as modificações relacionadas tenham sido feitas.

Se for especificado o nível de isolamento, ou o modo de leitura/escrita, a nova transação possuirá estas características, como se `SET TRANSACTION` tivesse sido executado.

## Parâmetros

WORK

TRANSACTION

Palavras chave opcionais. Não produzem nenhum efeito.

Consulte `SET TRANSACTION` para obter informações sobre o significado dos outros parâmetros deste comando.

## Observações

`START TRANSACTION` possui a mesma funcionalidade do `BEGIN`.

Use `COMMIT` ou `ROLLBACK` para terminar um bloco de transação.

Executar o `BEGIN` dentro de um bloco de transação provoca uma mensagem de advertência. O estado da transação não é afetado. Para aninhar transações dentro de um bloco de transação use pontos de salvamento (consulte `SAVEPOINT`).

Por motivo de compatibilidade podem ser omitidas as vírgulas entre *transaction\_modes* sucessivos.

## Exemplos

Para iniciar um bloco de transação:

```
BEGIN;
```

## Compatibilidade

O comando `BEGIN` é uma extensão do PostgreSQL à linguagem. É equivalente ao comando `START TRANSACTION` do padrão SQL, que deve ser visto para obter informações adicionais sobre compatibilidade.

Por coincidência, a palavra chave `BEGIN` é utilizada com uma finalidade diferente na linguagem SQL incorporada. Deve ser tomado muito cuidado com relação à semântica da transação ao migrar aplicativos de banco de dados.

**Consulte também**

*COMMIT, ROLLBACK, START TRANSACTION, SAVEPOINT*

# CHECKPOINT

## Nome

CHECKPOINT — força um ponto de controle no registro de transação

## Sinopse

CHECKPOINT

## Descrição

É colocado, periodicamente, um ponto de controle (*checkpoint*) no *registro prévio da escrita* (WAL = *write ahead logging*); para ajustar o intervalo do ponto de controle automático devem ser consultados os parâmetros de configuração em tempo de execução `checkpoint_segments` e `checkpoint_timeout`. Ao ser executado, o comando `CHECKPOINT` força um ponto de controle imediato, sem aguardar o ponto de controle agendado.

Um ponto de controle é um ponto na sequência de registros de transação onde todos os arquivos de dados são atualizados para refletir as informações registradas. Todos os dados são descarregados (*flushed*) no disco. Para obter informações adicionais sobre o sistema WAL deve ser consultado o Capítulo 25.

Somente os superusuários podem executar o comando `CHECKPOINT`. Este comando não foi feito para ser utilizado durante a operação normal.

## Compatibilidade

O comando `CHECKPOINT` é uma extensão do PostgreSQL à linguagem.

# CLOSE

## Nome

CLOSE — fecha o cursor

## Sinopse

CLOSE *nome*

## Descrição

O comando `CLOSE` libera os recursos associados a um cursor aberto. Após o cursor ser fechado, não é permitida nenhuma operação posterior que o utilize. O cursor deve ser fechado quando não for mais necessário.

Todo cursor aberto sem a cláusula `HOLD` é fechado implicitamente quando a transação termina por um `COMMIT` ou `ROLLBACK`. O cursor aberto com a cláusula `HOLD` é fechado implicitamente quando a transação em que foi criado é interrompida através de `ROLLBACK`. Se a transação que o criou for efetivada com sucesso, o cursor aberto com a cláusula `HOLD` permanece aberto até o `CLOSE` explícito ser executado, ou o cliente desconectar.

## Parâmetros

*nome*

O nome do cursor aberto a ser fechado.

## Observações

O PostgreSQL não possui um comando `OPEN` cursor explícito; o cursor é considerado aberto ao ser declarado. Use o comando `DECLARE` para declarar o cursor.

## Exemplos

Fechar o cursor `cur_emp`:

```
CLOSE cur_emp;
```

## Compatibilidade

O comando `CLOSE` está em conformidade total com o padrão SQL.

## Consulte também

*DECLARE*, *FETCH*, *MOVE*

# CLUSTER

## Nome

CLUSTER — agrupa a tabela de acordo com um índice

## Sinopse

```
CLUSTER nome_do_índice ON nome_da_tabela
CLUSTER nome_da_tabela
CLUSTER
```

## Descrição

O comando `CLUSTER` instrui o PostgreSQL para agrupar a tabela especificada por *nome\_da\_tabela*, com base no índice especificado por *nome\_do\_índice*. O índice deve ter sido definido anteriormente na tabela *nome\_da\_tabela*.

Ao ser agrupada, a tabela é fisicamente reordenada com base na informação do índice. O agrupamento é feito uma única vez: ao ser posteriormente atualizada, as modificações feitas na tabela não são agrupadas, ou seja, nenhuma tentativa é feita para manter as tuplas novas ou atualizadas na ordem do índice. Se for desejado, a tabela pode ser reagrupada periodicamente executando este comando novamente.

Quando a tabela é agrupada, o PostgreSQL registra qual foi o índice utilizado para agrupar. A forma `CLUSTER nome_da_tabela` reagrupa a tabela utilizando o mesmo índice utilizado anteriormente.

O comando `CLUSTER` sem nenhum parâmetro reagrupa todas as tabelas do banco de dados corrente pertencentes ao usuário que está executando o comando, ou todas as tabelas se for executado por um superusuário; as tabelas que nunca foram agrupadas não são incluídas. Esta forma do comando `CLUSTER` não pode ser chamada dentro de uma transação ou de uma função.

Durante o agrupamento da tabela é obtido o bloqueio `ACCESS EXCLUSIVE`, que não permite realizar qualquer outra operação de banco de dados na tabela, tanto de leitura quanto de escrita, até o comando `CLUSTER` terminar.

## Parâmetros

*nome\_do\_índice*

O nome do índice.

*nome\_da\_tabela*

O nome da tabela (opcionalmente qualificado pelo esquema).

## Observações

No caso do acesso aleatório a uma única linha da tabela, a ordem física dos dados na tabela não é importante. Entretanto, havendo tendência para acessar alguns dados mais que outros, e havendo um índice agrupando estes dados, a utilização do comando `CLUSTER` trará benefícios. Se for acessada uma faixa de valores indexados de uma tabela, ou um único valor indexado possuindo várias linhas correspondendo a este valor, o comando `CLUSTER` ajuda, porque quando o índice identifica a página da primeira linha, provavelmente todas as outras linhas estarão nesta mesma página, reduzindo o acesso ao disco e acelerando a consulta.

Durante a operação de agrupamento, uma cópia temporária da tabela é criada contendo os dados da tabela na ordem do índice. Também são criadas cópias temporárias de cada índice da tabela. Portanto, é necessário um espaço livre em disco pelo menos igual à soma do tamanho da tabela com os tamanhos de seus índices.

Como o comando `CLUSTER` guarda a informação de agrupamento, é possível agrupar as tabelas desejadas manualmente na primeira vez e, depois, configurar um evento periódico, como no `VACUUM`, para que as tabelas sejam periodicamente reagrupadas.

Como o planejador registra estatísticas sobre a ordem das linhas nas tabelas, é aconselhável executar o comando `ANALYZE` na tabela recém agrupada, senão o otimizador poderá fazer escolhas ruins no planejamento dos comandos.



Existe outra forma de agrupar os dados. O comando `CLUSTER` reordena a tabela original usando a ordem do índice especificado. Este procedimento pode ser lento para tabelas grandes, porque as linhas são lidas no disco na ordem do índice e, se a tabela não estiver ordenada, as linhas estarão em páginas aleatórias, fazendo uma página do disco ser lida para cada linha movida; o PostgreSQL possui um `cache`, mas a maioria das tabelas grandes não cabem no `cache`. A outra forma de agrupar a tabela é usar

```
CREATE TABLE nova_tabela AS
    SELECT lista_de_colunas FROM tabela ORDER BY lista_de_colunas;
```

que utiliza o código de ordenação da cláusula `ORDER BY` do PostgreSQL para criar a ordem desejada; geralmente é muito mais rápido que a varredura do índice para dados não ordenados. Em seguida, a tabela original deve ser removida, o comando `ALTER TABLE...RENAME` deve ser utilizado para mudar o nome da `nova_tabela` para o nome da tabela original, e recriados os índices da tabela. Entretanto, esta abordagem não preserva os OIDs, restrições, relacionamentos de chaves estrangeiras, privilégios concedidos e outras propriedades da tabela — todos estes itens deverão ser recriados manualmente.

## Exemplos

Agrupar a tabela `empregados` baseado no seu índice `idx_emp`:

```
CLUSTER idx_emp ON empregados;
```

Agrupar a tabela `empregados` utilizando o mesmo índice especificado anteriormente:

```
CLUSTER empregados;
```

Agrupar todas as tabelas de um banco de dados que foram agrupadas anteriormente:

```
CLUSTER;
```

## Compatibilidade

Não existe o comando `CLUSTER` no padrão SQL.

## Consulte também

*clusterdb*

# COMMENT

## Nome

COMMENT — define ou muda o comentário sobre um objeto

## Sinopse

```
COMMENT ON
{
    TABLE nome_do_objeto |
    COLUMN nome_da_tabela.nome_da_coluna |
    AGGREGATE nome_da_agregação (tipo_da_agregação) |
    CAST (tipo_de_origem AS tipo_de_destino) |
    CONSTRAINT nome_da_restrição ON nome_da_tabela |
    CONVERSION nome_do_objeto |
    DATABASE nome_do_objeto |
    DOMAIN nome_do_objeto |
    FUNCTION nome_da_função (tipo_do_argumento1, tipo_do_argumento2, ...) |
    INDEX nome_do_objeto |
    LARGE OBJECT oid_do_objeto_grande |
    OPERATOR nome_do_operador (tipo_do_operando_à_esquerda, tipo_do_operando_à_direita) |
    OPERATOR CLASS nome_do_objeto USING método_de_índice |
    [ PROCEDURAL ] LANGUAGE nome_do_objeto |
    RULE nome_da_regra ON nome_da_tabela |
    SCHEMA nome_do_objeto |
    SEQUENCE nome_do_objeto |
    TRIGGER nome_do_gatilho ON nome_da_tabela |
    TYPE nome_do_objeto |
    VIEW nome_do_objeto
} IS 'text'
```

## Descrição

O comando COMMENT armazena um comentário sobre um objeto do banco de dados.

Para modificar um comentário deve ser executado um novo comando COMMENT para o mesmo objeto. Somente é armazenada uma cadeia de caracteres contendo o comentário para cada objeto. Para remover o comentário deve ser escrito NULL no lugar da cadeia de caracteres. Os comentários são removidos automaticamente quando o objeto é removido.

Os comentários podem ser facilmente acessados através dos comandos \dd, \d+ e \l+ do psql. Outras interfaces de usuário podem acessar os comentários utilizando as mesmas funções nativas usadas pelo psql, ou seja: obj\_description() e col\_description() (Consulte a Tabela 9-45).

## Parâmetros

*nome\_do\_objeto*  
*nome\_da\_tabela.nome\_da\_coluna*  
*nome\_da\_agregação*  
*nome\_da\_restrição*  
*nome\_da\_função*  
*nome\_do\_operador*  
*nome\_da\_regra*  
*nome\_do\_gatilho*

O nome do objeto ao qual o comentário se refere. Os nomes das tabelas, agregações, domínios, funções, índices, operadores, classes de operador, seqüências, tipos e visões podem ser qualificados pelo esquema.

*tipo\_da\_agregação*

O tipo de dado do argumento da função de agregação, ou \* se a função aceitar qualquer tipo de dado.

*oid\_do\_objeto\_grande*

O identificador de objeto (OID) do objeto grande.

PROCEDURAL

Meramente informativo.

*tipo\_de\_origem*

O nome do tipo de dado de origem da transformação.

*tipo\_de\_destino*

O nome do tipo de dado de destino da transformação.

*text*

O novo comentário, escrito como um literal cadeia de caracteres; ou NULL para remover o comentário.

## Observações

Um comentário sobre o banco de dados somente pode ser criado no próprio banco de dados, e somente é visível neste banco de dados, e não nos demais bancos de dados.

Não existe atualmente nenhum mecanismo de segurança para os comentários: qualquer usuário conectado ao banco de dados pode ver todos os comentários sobre os objetos neste banco de dados; porém, somente os superusuários podem modificar comentários sobre objetos que não lhe pertencem. Portanto, não devem ser colocadas informações críticas para a segurança nos comentários.

## Exemplos

Anexar um comentário à tabela *minha\_tabela*:

```
COMMENT ON TABLE minha_tabela IS 'Esta tabela é minha.';
```

Remover o comentário:

```
COMMENT ON TABLE minha_tabela IS NULL;
```

Alguns outros exemplos:

```
COMMENT ON AGGREGATE minha_agregacao (double precision) IS 'Calcula a variância da amostra';
COMMENT ON CAST (text AS int4) IS 'Permite transformar texto em int4';
COMMENT ON COLUMN minha_tabela.minha_coluna IS 'Número de identificação do empregado';1
COMMENT ON CONVERSION minha_conversao IS 'Conversão para Unicode';
COMMENT ON DATABASE meu_bd IS 'Banco de dados de desenvolvimento';
COMMENT ON DOMAIN meu dominio IS 'Domínio de endereço de correio eletrônico';
COMMENT ON FUNCTION minha_funcao (timestamp) IS 'Retorna algarismos romanos';
COMMENT ON INDEX meu_indice IS 'Impõe a unicidade do identificador do empregado';
COMMENT ON LANGUAGE plpython IS 'Suporte a Python nos procedimentos armazenados';
COMMENT ON LARGE OBJECT 346344 IS 'Documento de planejamento';
COMMENT ON OPERATOR ^ (text, text) IS 'Realiza a interseção de dois textos';
COMMENT ON OPERATOR ^ (NONE, text) IS 'Este é um operador de prefixo para texto';
COMMENT ON OPERATOR CLASS int4ops USING btree IS 'Operadores inteiro de 4 bytes para árvores-B';
COMMENT ON RULE minha_regra ON minha_tabela IS 'Registra as atualizações das linhas dos empregados';
COMMENT ON SCHEMA meu_esquema IS 'Dados departamentais';
COMMENT ON SEQUENCE minha_sequencia IS 'Usado para gerar as chaves primárias';
COMMENT ON TABLE meu_esquema.minha_tabela IS 'Informações dos empregados';
COMMENT ON TRIGGER meu_gatilho ON minha_tabela IS 'Usado para integridade referencial';
COMMENT ON TYPE complex IS 'Tipo de dado de número complexo';
COMMENT ON VIEW minha_visao IS 'Visão dos custos departamentais';
```

## Compatibilidade

Não existe o comando `COMMENT` no padrão SQL.

## Notas

1. Também pode ser usada a forma `COMMENT ON COLUMN minha_visao.minha_coluna IS 'Comentário sobre a coluna da visão';` (N. do T.)

# COMMIT

## Nome

COMMIT — efetiva a transação corrente

## Sinopse

```
COMMIT [ WORK | TRANSACTION ]
```

## Descrição

O comando `COMMIT` efetiva a transação corrente. Todas as modificações efetuadas pela transação se tornam visíveis para os outros, e existe a garantia de permanecerem se uma falha ocorrer.

## Parâmetros

`WORK`

`TRANSACTION`

Palavras chave opcionais. Não produzem nenhum efeito.

## Observações

Use o *ROLLBACK* para interromper a transação.

A utilização do `COMMIT` fora de uma transação não causa nenhum problema, mas provoca uma mensagem de advertência.

## Exemplos

Para efetivar a transação corrente e tornar todas as mudanças permanentes:

```
COMMIT;
```

## Compatibilidade

O padrão SQL somente especifica as duas formas `COMMIT` e `COMMIT WORK`. Fora isso está totalmente em conformidade.

## Consulte também

*BEGIN*, *ROLLBACK*

# COPY

## Nome

COPY — copia dados entre um arquivo e uma tabela

## Sinopse

```
COPY nome_da_tabela [ ( coluna [, ...] ) ]  
FROM { 'nome_do_arquivo' | STDIN }  
[ [ WITH ]  
    [ BINARY ]  
    [ OIDS ]  
    [ DELIMITER [ AS ] 'delimitador' ]  
    [ NULL [ AS ] 'cadeia_de_caracteres_nula' ]  
    [ CSV [ QUOTE [ AS ] 'citação' ]  
        [ ESCAPE [ AS ] 'escape' ]  
        [ FORCE NOT NULL coluna [, ...] ]
```

```
COPY nome_da_tabela [ ( coluna [, ...] ) ]  
TO { 'nome_do_arquivo' | STDOUT }  
[ [ WITH ]  
    [ BINARY ]  
    [ OIDS ]  
    [ DELIMITER [ AS ] 'delimitador' ]  
    [ NULL [ AS ] 'cadeia_de_caracteres_nula' ]  
    [ CSV [ QUOTE [ AS ] 'citação' ]  
        [ ESCAPE [ AS ] 'escape' ]  
        [ FORCE QUOTE coluna [, ...] ]
```

## Descrição

O comando `COPY` copia dados entre tabelas do PostgreSQL e arquivos do sistema operacional. O comando `COPY TO` copia o conteúdo de uma tabela *para* um arquivo, enquanto o comando `COPY FROM` copia dados *de* um arquivo para uma tabela (adicionando os dados aos já existentes na tabela).

Se uma lista de colunas for especificada, o comando `COPY` somente copia os dados das colunas especificadas de/para o arquivo. Havendo colunas na tabela que não estejam na lista de colunas, o comando `COPY FROM` insere o valor padrão destas colunas.

O comando `COPY` com um nome de arquivo instrui o servidor PostgreSQL a ler ou escrever diretamente no arquivo. O arquivo deve ser acessível ao servidor, e o nome deve ser especificado sob o ponto de vista do servidor. Quando `STDIN` ou `STDOUT` são especificados, os dados são transmitidos através da conexão entre o cliente e o servidor.

## Parâmetros

*nome\_da\_tabela*

O nome de uma tabela existente (opcionalmente qualificado pelo esquema).

*coluna*

A lista opcional das colunas a serem copiadas. Se nenhuma lista for especificada, todas as colunas são utilizadas.

*nome\_do\_arquivo*

O nome do caminho absoluto do arquivo de entrada ou de saída.

`STDIN`

Especifica que a entrada vem do aplicativo cliente.

**STDOUT**

Especifica que a saída vai para o aplicativo cliente.

**BINARY**

Faz todos os dados serem armazenados ou lidos no formato binário, em vez de texto. Não é possível especificar as opções `DELIMITER`, `NULL` ou `CSV` no modo binário.

**oids**

Especifica que deve ser copiado o identificador interno do objeto (OID) de cada linha; é gerado um erro se `oids` for especificado para uma tabela que não possua OIDs.

*delimiter*

O caractere único que separa as colunas dentro de cada linha do arquivo. O padrão é o caractere de tabulação no modo texto, e a vírgula no modo `CSV`.

*cadeia\_de\_caracteres\_nula*

A cadeia de caracteres que representa o valor nulo. O padrão é “\N” (contrabarra-N) no modo texto, e um valor vazio sem os caracteres de citação (aspas, por padrão) no modo `CSV`. Pode-se preferir a cadeia de caracteres vazia, mesmo no modo texto, se não for desejado fazer distinção entre nulos cadeias de caracteres vazias.

**Nota:** No `COPY FROM` qualquer item de dado correspondendo a esta cadeia de caracteres é armazenado com o valor nulo e, portanto, deve haver certeza que está sendo utilizada a mesma cadeia de caracteres utilizada para fazer o `COPY TO`.

**CSV**

Seleciona o modo valor separado por vírgula (`CSV`).

*citação*

Especifica o caractere de citação (`quotation character`) no modo `CSV`. Aspas por padrão.

*escape*

Especifica o caractere que deve preceder o valor do caractere de dado `QUOTE` no modo `CSV`. O padrão é o mesmo valor de `QUOTE` (geralmente aspas).

**FORCE QUOTE**

No modo `COPY TO` do `CSV` força a utilização do caractere de citação (aspas, por padrão) em todos os valores diferentes de `NULL` em cada uma das colunas especificadas. A saída `NULL` nunca é colocada entre os caracteres de citação.

**FORCE NOT NULL**

No modo `COPY TO` do `CSV` processa cada coluna especificada como se estivesse entre os caracteres de citação (aspas, por padrão) e, portanto, não sendo um valor `NULL`. Para a cadeia de caracteres nula padrão no modo `CSV` ( ' '), faz com que os valores faltando sejam entrados como cadeias de caracteres de comprimento zero.

**Observações**

O comando `COPY` só pode ser utilizado em tabelas, não podendo ser utilizado em visões.

A palavra chave `BINARY` faz todos os dados serem armazenados/lidos no formato binário em vez de texto. É um pouco mais rápido que o modo texto normal, mas o arquivo produzido no formato binário é menos portátil entre arquiteturas de máquinas e versões do PostgreSQL.

É necessário possuir o privilégio de seleção na tabela cujos valores são lidos pelo `COPY TO`, e o privilégio inserção na tabela onde os valores são inseridos pelo `COPY FROM`.

Os arquivos declarados no comando `COPY` são lidos ou escritos diretamente pelo servidor, e não pelo aplicativo cliente. Portanto, devem residir, ou serem acessíveis, pela máquina servidora de banco de dados, e não pela estação cliente. Os arquivos devem ser acessíveis e poderem ser lidos ou escritos pelo usuário do PostgreSQL (o ID do usuário sob o qual o servidor executa), e não pelo cliente. O `COPY` com nome de arquivo só é permitido aos superusuários do banco de dados, porque permite ler e escrever em qualquer arquivo que o servidor possua privilégio de acesso.

Não confunda o comando `COPY` com a instrução `\copy` do `psql`. O `\copy` executa `COPY FROM STDIN` ou `COPY TO STDOUT` e, portanto, lê/grava os dados em um arquivo acessível ao cliente `psql`. Por esta razão, a acessibilidade e os direitos de acesso ao arquivo dependem do cliente, e não do servidor, quando o `\copy` é utilizado.

Recomenda-se que o nome do arquivo utilizado no comando `COPY` seja sempre especificado como um caminho absoluto, o que é exigido pelo servidor no caso do `COPY TO`, mas para o `COPY FROM` existe a opção de ler um arquivo especificado pelo caminho relativo. O caminho é interpretado com relação ao diretório de trabalho do processo servidor (algum lugar abaixo do diretório de dados), e não relativo ao diretório de trabalho do cliente.

O comando `COPY FROM` chama os gatilhos e as restrições de verificação da tabela de destino. Entretanto, não chama as regras.

A entrada e a saída do `COPY` são afetadas por `DateStyle`. Para garantir a portabilidade com outras instalações do PostgreSQL, que podem utilizar definições para `DateStyle` diferentes do padrão, `DateStyle` deve ser definida como `ISO` antes de usar `COPY TO`.

O `COPY` pára de executar no primeiro erro, o que não deve causar problemas no caso do `COPY TO`, mas a tabela de destino já terá recebido as primeiras linhas no caso do `COPY FROM`. Estas linhas não são visíveis nem acessíveis, mas ainda assim ocupam espaço em disco, podendo causar o desperdício de uma quantidade considerável de espaço em disco, se o erro ocorrer durante a cópia de uma grande quantidade de dados. Deve ser executado o comando `VACUUM` para recuperar o espaço desperdiçado.

## Formatos dos arquivos

### Formato texto

Quando o comando `COPY` é utilizado sem as opções `BINARY` ou `CSV`, os dados são lidos ou escritos em um arquivo texto com uma linha para cada linha da tabela. As colunas de cada linha são separadas pelo caractere delimitador. Os valores das colunas são cadeias de caracteres geradas pela função de saída, ou aceitas pela função de entrada, do tipo de dado de cada atributo. A cadeia de caracteres nula especificada é utilizada no lugar das colunas que são nulas. O comando `COPY FROM` produz um erro se alguma linha do arquivo de entrada possuir mais, ou menos, colunas que o esperado. Se `OIDS` for especificado, o `OID` é lido ou escrito como a primeira coluna, antecedendo as colunas de dado do usuário.

O fim dos dados pode ser representado por uma única linha contendo apenas contrabarra-ponto (`\.`). A marca de fim-dados não é necessária ao ler de um arquivo, porque o fim-de-arquivo serve perfeitamente bem; é necessária apenas ao copiar dados de/para aplicativos cliente quando for utilizado um protocolo cliente anterior ao 3.0.

Caracteres contrabarra (`\`) podem ser utilizados nos dados do comando `COPY` para evitar que caracteres dos dados sejam interpretados como delimitadores de linha ou de coluna. Em particular, os seguintes caracteres *devem* ser precedidos por uma contrabarra se fizerem parte do valor de uma coluna: a própria contrabarra, a nova-linha (LF), o retorno-do-carro (CR) e o caractere delimitador corrente.

A cadeia de caracteres nula é enviada pelo `COPY TO` sem adição de contrabarras; inversamente, o `COPY FROM` verifica a entrada com relação à cadeia de caracteres nula antes de remover as contrabarras. Portanto, uma cadeia de caracteres nula como o `\N` não pode ser confundida com o valor de dado `\N` (que seria representado por `\\N`).

As seguintes seqüências especiais de contrabarra são reconhecidas pelo comando `COPY FROM`:

Seqüência	Representa
<code>\b</code>	Retorna apagando (Backspace) (ASCII 8)
<code>\f</code>	Avanço do formulário (Form feed) (ASCII 12)
<code>\n</code>	Nova-linha (Newline) (ASCII 10)
<code>\r</code>	Retorno do carro (Carriage return) (ASCII 13)
<code>\t</code>	Tabulação (ASCII 9)
<code>\v</code>	Tabulação vertical (ASCII 11)
<code>\dígitos</code>	A contrabarra seguida por um a três dígitos octais especifica o caractere com este código numérico



Atualmente o `COPY TO` não produz seqüências do tipo contrabarra dígitos-octais, mas utiliza as outras seqüências de contrabarra listadas acima para estes caracteres de controle.

Qualquer outro caractere precedido por contrabarra que não tenha sido mencionado na tabela acima é interpretado como representando a si próprio. Entretanto, tome cuidado ao adicionar contrabarras desnecessariamente uma vez que isto poderá, acidentalmente, produzir uma cadeia de caracteres correspondendo à marca de fim-de-dados (`\.`), ou a cadeia de caracteres nula (`\N` por padrão). Estas cadeias de caracteres são reconhecidas antes de ocorrer qualquer outro processamento da contrabarra.

É altamente recomendado que os aplicativos que geram dados para o `COPY` convertam os caracteres de nova-linha e de retorno-de-carro presentes nos dados nas seqüências `\n` e `\r`, respectivamente. Atualmente é possível representar retorno-de-carro nos dados por contrabarra e retorno-de-carro, e representar nova-linha nos dados por contrabarra e nova-linha. Entretanto, estas representações podem não ser aceitas nas versões futuras. São, também, altamente vulneráveis à corrupção quando o arquivo do `COPY` é transferido entre máquinas diferentes; por exemplo, do Unix para o Windows, e vice-versa.

O comando `COPY TO` termina cada linha pelo caractere de nova-linha ("`\n`"), no estilo Unix. Os servidores executando no Microsoft Windows em vez disto geram retorno-de-carro/nova-linha ("`\r\n`"), mas somente no `COPY` para um arquivo no servidor; para manter a consistência entre as plataformas, `COPY TO STDOUT` sempre gera "`\n`", independentemente da plataforma do servidor. O comando `COPY FROM` pode tratar linhas terminando por nova-linha, retorno-de-carro, ou retorno-de-carro/nova-linha. Para reduzir o risco de erro devido a caracteres de nova-linha ou de retorno-de-carro sem contrabarra que fazem parte dos dados, o `COPY FROM` reclama se o final de todas as linhas de entrada não forem idênticos.

## Formato CSV

Este formato é utilizado para importar e exportar arquivos no formato Valor Separado por Vírgula (*Comma Separated Value* – CSV) usado por vários outros programas, como as planilhas eletrônicas. Em vez de utilizar o escape padrão do modo texto do PostgreSQL, gera e reconhece o mecanismo de escape comum do CSV.

Em cada registro os valores são separados pelo caractere `DELIMITER`. Se o valor contiver o caractere delimitador, o caractere `QUOTE`, a cadeia de caracteres `NULL`, um caractere de retorno-de-carro ou de nova-linha, então todo o valor recebe como prefixo e sufixo o caractere `QUOTE`, e qualquer ocorrência do caractere `QUOTE` ou do caractere `ESCAPE` dentro do valor é precedida pelo caractere de escape. Também pode ser utilizado `FORCE QUOTE` para obrigar a colocar o caractere `QUOTE` nos valores diferentes de `NULL` em determinadas colunas.

O formato CSV não possui uma forma padronizada para distinguir entre o valor `NULL` e uma cadeia de caracteres vazia. O `COPY` do PostgreSQL trata isto através de citações (aspas). O valor `NULL` é escrito como `NULL` sem caracteres de citação, enquanto o valor de dado correspondendo à cadeia de caracteres `NULL` é escrito entre caracteres de citação. Portanto, usando a configuração padrão, o valor `NULL` é escrito como uma cadeia de caracteres vazia sem os caracteres de citação, enquanto uma cadeia de caracteres vazia é escrita entre aspas ("`"`"). A leitura dos valores segue regras semelhantes. Pode ser utilizado `FORCE NOT NULL` para inibir na entrada comparações de `NULL` para determinadas colunas.

**Nota:** O modo CSV tanto reconhece quanto gera arquivos CSV com valores entre caracteres de citação contendo retorno-de-carro e nova-linha embutidos. Portanto, os arquivos não são exatamente uma linha para cada linha da tabela como nos arquivos do modo texto. Entretanto, o PostgreSQL rejeita a entrada para o `COPY` se houver embutida em algum campo uma seqüência de caracteres de fim de linha que não corresponda à convenção de fim de linha usada no próprio arquivo CSV. Geralmente é mais seguro importar dados contendo caracteres de fim de linha usando os formatos texto ou binário em vez do CSV.

**Nota:** Muitos programas produzem arquivos CSV estranhos e ocasionalmente maldosos; portanto, este formato de arquivo é mais uma convenção do que um padrão. Por isso, podem ser encontrados arquivos que não podem ser importados utilizando este mecanismo, e o `COPY` pode produzir arquivos que outros programas não consigam processar.

## Formato Binário

O formato do arquivo usado pelo `COPY BINARY` mudou no PostgreSQL v7.4. O novo formato consiste em um cabeçalho do arquivo, zero ou mais tuplas contendo os dados das linhas, e um rodapé do arquivo. Os cabeçalhos e os dados agora são enviados na ordem de byte da rede.

### Cabeçalho do Arquivo

O cabeçalho do arquivo é formado por 15 bytes para campos fixos, seguidos por uma área de extensão do cabeçalho de comprimento variável. Os campos fixos são:

## Assinatura

A sequência de 11 bytes `PGCOPY\0377\0` — observe que o byte zero é uma parte requerida da assinatura (A assinatura foi projetada para permitir a fácil identificação de arquivos corrompidos por uma transferência de dados não apropriada. Esta assinatura é modificada por filtros de tradução de fim de linha, bytes zero suprimidos, bits altos suprimidos, ou mudanças de paridade).

## Campo de sinalizadores

Inteiro de 32 bits máscara de bits, indicando aspectos importantes do formato do arquivo. Os bits são numerados de 0 (LSB) a 31 (MSB). Deve ser observado que este campo é armazenado na ordem de bytes da rede (byte mais significativo primeiro), assim como todos os campos inteiro utilizados no formato do arquivo. Os bits 16-31 são reservados para indicar questões críticas do formato do arquivo; a leitura deve ser interrompida se for encontrado neste intervalo um bit definido não esperado. Os bits 0-15 são reservados para sinalizar questões de formato anteriores-compatíveis; a leitura deve simplesmente ignorar qualquer bit definido não esperado neste intervalo. Atualmente somente está definido um bit sinalizador, os demais devem ser zero:

### Bit 16

Se for 1, os OIDs estão incluídos nos dados; se for 0, não.

## Comprimento da área de extensão do cabeçalho

Inteiro de 32 bits, contendo o comprimento em bytes do restante do cabeçalho, não se incluindo. Atualmente é igual a zero, e a primeira tupla o segue imediatamente. Mudanças futuras no formato poderão permitir dados adicionais estarem presentes no cabeçalho. A leitura deve simplesmente pular qualquer dado na extensão do cabeçalho que não souber o que fazer com o mesmo.

A área de extensão do cabeçalho foi concebida para conter uma sequência de blocos auto-identificadores. O campo de sinalizadores não tem por finalidade informar aos leitores o que existe na área de extensão. O projeto específico do conteúdo da extensão do cabeçalho foi deixado para uma versão futura.

Este projeto permite tanto adições de cabeçalhos compatíveis com os anteriores (adicionar blocos de extensão de cabeçalho, ou definir bits sinalizadores de baixa-ordem), quanto mudanças não compatíveis com os anteriores (definir bits sinalizadores de alta-ordem para sinalizar estas mudanças, e adicionar dados de apoio à área de extensão se for necessário).

## Tuplas

Cada tupla começa por um inteiro de 16 bits, que é o contador do número de campos na tupla; atualmente todas as tuplas da tabela possuem o mesmo contador, mas isto pode não ser verdade para sempre. Então, para cada campo da tupla, existe a informação do comprimento com 32 bits, seguida por esta quantidade de bytes de dados do campo; a informação do comprimento não se inclui, podendo ser zero. Como caso especial, -1 informa o valor de um campo nulo, e nenhum byte de valor vem a seguir.

Não existe nenhum enchimento de alinhamento ou qualquer outro dado adicional entre os campos.

Atualmente é assumido que todos os valores dos dados em um arquivo `COPY BINARY` estão no formato binário (código de formatação um). É previsto que uma extensão futura poderá adicionar um campo de cabeçalho permitindo que os códigos de formatação sejam especificados por coluna.

Para determinar o formato binário apropriado para os dados da tupla deve ser consultado o código fonte do PostgreSQL, em particular as funções `*send` e `*recv` para o tipo de dado de cada coluna (normalmente estas funções se encontram no diretório `src/backend/utils/adt/` da distribuição do código fonte).

Se os OIDs forem incluídos no arquivo, o campo OID segue imediatamente a informação contador do número de campos. É um campo normal, exceto que não está incluído no contador do número de campos. Em particular possui uma informação de comprimento — permitindo tratar OIDs de 4-bytes versus 8-bytes sem muita dificuldade e, também, permitindo os OIDs serem mostrados como nulo se por acaso for desejado.

## Rodapé do Arquivo

O rodapé do arquivo consiste de um inteiro de 16-bits contendo -1. É facilmente distinguível do contador do número de campos da tupla.

A leitura deve relatar um erro se a informação contador do número de campos não for -1 nem for o número esperado de colunas. Isto permite uma verificação adicional com relação à perda de sincronização com os dados.

## Exemplos

O exemplo a seguir copia uma tabela para o cliente utilizando a barra vertical (|) como delimitador de campo:

```
COPY países TO STDOUT WITH DELIMITER '|' ;
```

Para copiar os dados de um arquivo para a tabela países:

```
COPY países FROM '/usr1/proj/bray/sql/dados_dos_países' ;
```

Abaixo está mostrado um exemplo contendo dados apropriados para serem copiados a partir da STDIN:

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
```

Deve ser observado que em cada linha o espaço em branco é, na verdade, o caractere de tabulação.

Abaixo estão os mesmos dados escritos no formato binário. Os dados mostrados foram filtrados utilizando o utilitário do Unix `od -c`. A tabela possui três colunas: a primeira é do tipo `char(2)`; a segunda é do tipo `text`; a terceira é do tipo `integer`. Todas as linhas possuem o valor nulo na terceira coluna.

```
0000000  P  G  C  O  P  Y  \n 377  \r  \n  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002  A  F  \0  \0  \0 013  A
0000040  F  G  H  A  N  I  S  T  A  N 377 377 377 377  \0 003
0000060  \0  \0  \0 002  A  L  \0  \0  \0 007  A  L  B  A  N  I
0000100  A 377 377 377 377  \0 003  \0  \0  \0 002  D  Z  \0  \0  \0
0000120 007  A  L  G  E  R  I  A 377 377 377 377  \0 003  \0  \0
0000140  \0 002  Z  M  \0  \0  \0 006  Z  A  M  B  I  A 377 377
0000160 377 377  \0 003  \0  \0  \0 002  Z  W  \0  \0  \0  \b  Z  I
0000200  M  B  A  B  W  E 377 377 377 377 377 377
```

No próximo exemplo <sup>1</sup> é utilizado o comando `COPY`, chamado a partir do `psql`, para copiar a primeira, a segunda e a quarta colunas de uma tabela de quatro colunas, a partir da quarta linha do arquivo `/tmp/linhas.dat`. A tabela foi criada através do comando:

```
CREATE TABLE linhas (
coluna1 TEXT PRIMARY KEY,
coluna2 TEXT,
coluna3 TEXT,
coluna4 TEXT);
```

O conteúdo do arquivo carregado está mostrado abaixo:

```
# cat /tmp/linhas.dat
linha1a|linha1b|linha1d
linha2a|linha2b|linha2d
linha3a|linha3b|linha3d
linha4a|linha4b|linha4d
linha5a|linha5b|linha5d
linha6a|linha6b|linha6d
linha7a|linha7b|linha7d
linha8a|linha8b|linha8d
```

E o comando utilizado para copiar os dados do arquivo para a tabela foi:

```
tail +4 /tmp/linhas.dat | psql -U teste teste -c \
"COPY linhas(coluna1,coluna2,coluna4) FROM STDIN WITH DELIMITER '|' "
```

Como resultado os seguintes dados foram incluídos na tabela:

```
select * from linhas;
```

```

coluna1 | coluna2 | coluna3 | coluna4
-----+-----+-----+-----
linha4a | linha4b |         | linha4d
linha5a | linha5b |         | linha5d
linha6a | linha6b |         | linha6d
linha7a | linha7b |         | linha7d
linha8a | linha8b |         | linha8d
(5 linhas)

```

## Compatibilidade

Não existe o comando `COPY` no padrão SQL.

A sintaxe mostrada abaixo era utilizada nas versões anteriores a 7.3, sendo ainda aceita:

```

COPY [ BINARY ] nome_da_tabela [ WITH OIDS ]
FROM { 'nome_do_arquivo' | STDIN }
[ [USING] DELIMITERS 'delimitador' ]
[ WITH NULL AS 'cadeia_de_caracteres_nula' ]

```

```

COPY [ BINARY ] nome_da_tabela [ WITH OIDS ]
TO { 'nome_do_arquivo' | STDOUT }
[ [USING] DELIMITERS 'delimitador' ]
[ WITH NULL AS 'cadeia_de_caracteres_nula' ]

```

## Notas

1. Este exemplo foi escrito pelo tradutor, não fazendo parte do manual original.

# CREATE AGGREGATE

## Nome

CREATE AGGREGATE — cria uma função de agregação

## Sinopse

```
CREATE AGGREGATE nome (  
    BASETYPE = tipo_de_dado_da_entrada,  
    SFUNC = função_de_transição_de_estado,  
    STYPE = tipo_de_dado_do_estado  
    [ , FINALFUNC = função_final ]  
    [ , INITCOND = condição_inicial ]  
)
```

## Descrição

O comando `CREATE AGGREGATE` cria uma função de agregação. Algumas funções de agregação básicas e comumente utilizadas estão incluídas na distribuição; estão documentadas na Seção 9.15. Se forem criados tipos novos, ou se for necessária uma função de agregação não fornecida, então o comando `CREATE AGGREGATE` pode ser utilizado para fornecer as funcionalidades desejadas.

Se for fornecido o nome do esquema (por exemplo, `CREATE AGGREGATE meu_esquema.minha_agregacao ...`) então a função de agregação é criada no esquema especificado, senão é criada no esquema corrente.

Uma função de agregação é identificada pelo seu nome e tipo de dado de entrada. Duas funções de agregação no mesmo esquema podem ter o mesmo nome se operarem em tipos de dado de entrada diferentes. O nome e tipo de dado de entrada de uma função de agregação também deve ser diferente do nome e tipo(s) de dado de entrada de todas as funções comuns no mesmo esquema.

Uma função de agregação é composta por uma ou duas funções comuns: uma função de transição de estado, *função\_de\_transição\_de\_estado*, e uma função opcional para a realização dos cálculos finais, *função\_final*. Estas funções são utilizadas da seguinte forma:

```
função_de_transição_de_estado( estado_interno, próximo_item_de_dado ) --->  
próximo_estado_interno  
função_final( estado_interno ) ---> valor_da_agregação
```

O PostgreSQL cria uma variável temporária com o tipo de dado *tipo\_de\_dado\_do\_estado* para armazenar o estado interno corrente da agregação. Para cada item de dado da entrada a função de transição de estado é chamada para calcular o novo valor do estado interno. Após todos os dados terem sido processados, a função final é chamada uma vez para calcular o valor retornado da agregação. Não havendo nenhuma função final, então o valor do estado final é retornado como estiver.

A função de agregação pode fornecer uma condição inicial, ou seja, um valor inicial para o valor do estado interno. Este valor é especificado e armazenado no banco de dados em uma coluna do tipo `text`, mas deve possuir uma representação externa válida para uma constante do tipo de dado do valor do estado. Se não for fornecido, então o valor do estado começa com nulo.

Se a função de transição de estado for declarada como “strict”, então não poderá ser chamada com valores da entrada nulos. Para este tipo de função de transição, a execução da agregação se comporta da seguinte forma: Valores da entrada nulos são ignorados (a função não é chamada e o valor do estado anterior permanece); Se o valor do estado inicial for nulo, então o primeiro valor da entrada que não for nulo substitui o valor do estado, e a função de transição é chamada a partir do segundo valor da entrada que não for nulo. Este procedimento é útil para implementar funções de agregação como `max`. Deve ser observado que este comportamento somente está disponível quando o *tipo\_de\_dado\_do\_estado* for o mesmo do *tipo\_de\_dado\_da\_entrada*. Quando estes tipos de dado forem diferentes, deverá ser fornecido um valor não nulo para a condição inicial, ou utilizar uma função de transição que não seja estrita.

Se a função de transição de estado não for estrita então será chamada, incondicionalmente, para cada valor da entrada, devendo ser capaz de lidar com entradas nulas e valores de transição nulos por si própria. Esta opção permite ao autor da função de agregação ter pleno controle sobre o tratamento dos valores nulos.

Se a função final for declarada como “strict”, então não será chamada quando o valor do estado final for nulo; em vez disso, um resultado nulo será retornado automaticamente (É claro que este é apenas o comportamento normal de funções estritas <sup>1</sup>). A função final sempre tem a opção de retornar o valor nulo. Por exemplo, a função final para `avg` retorna nulo quando não há linhas de entrada.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) da função de agregação a ser criada.

*tipo\_de\_dado\_da\_entrada*

O tipo do dado de entrada sobre o qual esta função de agregação opera. Pode ser especificado como “ANY” para uma função de agregação que não examina seus valores de entrada (um exemplo é a função `count(*)`).

*função\_de\_transição\_de\_estado*

O nome da função de transição de estado a ser chamada para cada valor dos dados da entrada. Normalmente esta função possui dois argumentos, o primeiro sendo do tipo *tipo\_de\_dado\_do\_estado* e o segundo do tipo *tipo\_de\_dado\_da\_entrada*. Outra possibilidade, para funções de agregação que não examinam seus valores de entrada, é a função possuir apenas um argumento do tipo *tipo\_de\_dado\_do\_estado*. Em qualquer um dos casos a função deve retornar um valor do tipo *tipo\_de\_dado\_do\_estado*. Esta função recebe o valor do estado corrente e o item de dado da entrada corrente e retorna o próximo valor do estado.

*tipo\_de\_dado\_do\_estado*

O tipo de dado do valor do estado da agregação.

*função\_final*

O nome da função final chamada para calcular o resultado da agregação após todos os dados da entrada terem sido examinados. A função deve receber um único argumento do tipo *tipo\_de\_dado\_do\_estado*. O tipo de dado retornado pela agregação é definido pelo tipo retornado por esta função. Se a *função\_final* não for especificada, então o valor do estado final é utilizado como sendo o resultado da agregação, e o tipo retornado fica sendo o *tipo\_de\_dado\_do\_estado*.

*condição\_inicial*

A definição inicial do valor do estado. Deve ser uma constante cadeia de caracteres na forma aceita pelo tipo de dado *tipo\_de\_dado\_do\_estado*. Se não for especificado, o valor do estado começa com nulo.

Os parâmetros para `CREATE AGGREGATE` podem ser escritos em qualquer ordem, e não apenas na ordem mostrada acima.

## Exemplos

Consulte a Seção 31.10.

## Compatibilidade

O comando `CREATE AGGREGATE` é uma extensão do PostgreSQL à linguagem. O padrão SQL não inclui funções de agregação definidas pelo usuário.

## Consulte também

`ALTER AGGREGATE`, `DROP AGGREGATE`

## Notas

1. `strict` — A função *f* é estrita em um argumento se “*f* bottom = bottom”. Em outras palavras, o resultado depende do argumento *e*, portanto, a avaliação da aplicação da função não pode terminar enquanto a avaliação do argumento não

tenha terminado. FOLDOC - Free On-Line Dictionary of Computing  
(<http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=strict>) (N. do T.)

# CREATE CAST

## Nome

CREATE CAST — cria uma conversão de tipo de dado

## Sinopse

```
CREATE CAST (tipo_de_origem AS tipo_de_destino)
    WITH FUNCTION nome_da_função (tipos_dos_argumentos)
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

```
CREATE CAST (tipo_de_origem AS tipo_de_destino)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

## Descrição

O comando `CREATE CAST` cria uma conversão. A conversão especifica como realizar a conversão entre dois tipos de dado. Por exemplo,

```
SELECT CAST(42 AS text);
```

converte a constante inteira 42 para o tipo `text` chamando uma função especificada previamente, neste caso `text(int4)`; se nenhuma conversão adequada tiver sido definida, a conversão falha.

Dois tipos podem ser *binariamente compatíveis*, significando que podem ser convertidos um no outro “livremente”, sem chamar nenhuma função. Requer que os valores correspondentes utilizem a mesma representação interna. Por exemplo, os tipos `text` e `varchar` são binariamente compatíveis.

Por padrão, a conversão somente pode ser feita por uma solicitação de conversão explícita, ou seja, uma construção explícita `CAST(x AS nome_do_tipo)` ou `x::nome_do_tipo`.

Se a conversão for marcada como `AS ASSIGNMENT` então pode ser chamada implicitamente quando for atribuído um valor a uma coluna com o tipo de dado de destino. Por exemplo, supondo que `foo.f1` seja uma coluna do tipo `text`, então a atribuição

```
INSERT INTO foo (f1) VALUES (42);
```

é permitida se a conversão do tipo `integer` para o tipo `text` estiver marcada como `AS ASSIGNMENT`, senão a atribuição não é permitida; geralmente é utilizado o termo *conversão de atribuição* (`assignment cast`) para descrever este tipo de conversão.

Se a conversão estiver marcada como `AS IMPLICIT` então poderá ser chamada implicitamente em qualquer contexto, seja em uma atribuição ou internamente em uma expressão. Por exemplo, como `||` recebe operandos do tipo `text`, então a expressão

```
SELECT 'Data e hora ' || now();
```

somente será permitida se a conversão do tipo `timestamp` para o tipo `text` estiver marcada como `AS IMPLICIT`, senão será necessário escrever a conversão explicitamente como, por exemplo,

```
SELECT 'Data e hora ' || CAST(now() AS text);
```

(Geralmente é utilizado o termo *conversão implícita* (`implicit cast`) para descrever este tipo de conversão).

É sensato ser conservador com relação a marcar conversões como implícitas. Uma superabundância de possibilidades de conversões implícitas pode fazer o PostgreSQL escolher interpretações surpreendentes para os comandos, ou até não ser capaz de resolver o comando devido à existência de várias interpretações possíveis. Uma boa regra empírica é tornar a conversão chamável implicitamente somente nos casos de transformações que preservam as informações entre tipos da mesma categoria geral. Por exemplo, a conversão de `int2` em `int4` pode ser implícita, mas a conversão de `float8` para



`int4` provavelmente deve ser somente de atribuição. Conversões entre tipos de categorias diferentes, como `text` para `int4`, é melhor serem somente explícitas.

É necessário ser o dono do tipo de dado de origem ou de destino para poder criar uma conversão. Para criar uma conversão binariamente compatível é necessário ser um superusuário; esta restrição existe porque uma conversão binariamente compatível pode facilmente derrubar o servidor.

## Parâmetros

*tipo\_de\_origem*

O nome do tipo de dado de origem da conversão.

*tipo\_de\_destino*

O nome do tipo de dado de destino da conversão.

*nome\_da\_função(tipos\_dos\_argumentos)*

A função utilizada para realizar a conversão. O nome da função pode ser qualificado pelo esquema. Se não for, a função será procurada no caminho de procura de esquema. O tipo de dado do resultado da função deve corresponder ao tipo de dado de destino da conversão. Os argumentos são discutidos abaixo.

WITHOUT FUNCTION

Indica que o tipo de dado de origem e o tipo de dado de destino são binariamente compatíveis e, por isso, não é necessária nenhuma função para realizar a conversão.

AS ASSIGNMENT

Indica que a conversão pode ser chamada implicitamente em contextos de atribuição.

AS IMPLICIT

Indica que a conversão pode ser chamada implicitamente em qualquer contexto.

As funções de implementação de conversão podem ter de um a três argumentos. O tipo do primeiro argumento deve ser idêntico ao tipo de origem da conversão. O segundo argumento, se houver, deve ser do tipo `integer`; recebe o modificador de tipo associado ao tipo de destino, ou `-1` se não houver nenhum. O terceiro argumento, se houver, deve ser do tipo `boolean`; recebe `true` se a conversão for explícita, ou `false` caso contrário (Estranhamente, a especificação SQL obriga comportamentos diferentes para conversões implícitas e explícitas em alguns casos. Este argumento é fornecido para funções que devem implementar estes tipos de conversão. Não é recomendado que se projete tipos de dado próprios de uma forma que isto tenha importância).

Normalmente, a conversão deve ter tipos de dado de origem e de destino diferentes. Entretanto, é permitido declarar uma conversão com tipos de dados de origem e de destino idênticos, se houver uma função de implementação da conversão com mais de um argumento. É utilizado para representar nos catálogos do sistema funções de modificação do comprimento específicas do tipo (`type-specific length coercion functions`). A função nomeada é utilizada para modificar (`coerce`) um valor do tipo para o valor do modificador do tipo indicado por seu segundo argumento (uma vez que atualmente a gramática permite que apenas certos tipos de dados nativos possuam modificadores de tipo, esta funcionalidade não tem utilidade para os tipos de destino definidos pelo usuário, mas de qualquer forma é mencionado para ficar completo).

Quando uma conversão possui tipos de origem e de destino diferentes, e a função recebe mais de um argumento, representa a conversão de um tipo para o outro e a aplicação da modificação do comprimento em um único passo. Quando não existe disponível uma entrada deste tipo, a modificação (`coercion`) para um tipo que usa um modificador de tipo envolve duas etapas, uma para converter entre os tipos de dado e uma segunda para aplicar o modificador.

## Observações

Use o comando *DROP CAST* para remover conversões criadas pelo usuário.

Lembre-se que para ser possível converter tipos nas duas direções é necessário declarar conversões para as duas direções explicitamente.

Antes do PostgreSQL 7.3 toda função que possuía o mesmo nome de um tipo de dado, retornava este tipo de dado, e recebia um argumento de um tipo diferente era, automaticamente, uma função de conversão. Esta convenção foi

abandonada devido à introdução dos esquemas e da capacidade de representar conversões binariamente compatíveis nos catálogos do sistema. As funções de conversão internas ainda seguem este esquema de nomes, mas também devem aparecer como conversões no catálogo do sistema `pg_cast`.

Embora não seja requerido, é recomendado que se continue a seguir esta antiga convenção para dar nome às funções de implementação da conversão usando o nome do tipo de dado de destino. Muitos usuários estão acostumados a poderem converter tipos de dado usando a notação no estilo de função, ou seja, `nome_do_tipo(x)`. Na verdade esta notação não é nada mais, nem menos, que uma chamada à função de implementação da conversão; não é tratada especialmente como uma conversão. Se os nomes de suas funções de conversão não aderirem a esta convenção então seus usuários ficarão surpresos. Uma vez que o PostgreSQL permite a sobrecarga do mesmo nome de função com argumentos de tipos diferentes, não há dificuldade em ter-se várias funções de conversão de tipos diferentes todas usando o nome do tipo de destino.

**Nota:** Existe uma pequena mentira no parágrafo anterior: existe ainda um caso em que `pg_cast` é utilizada para resolver o significado de uma aparente chamada de função. Se a chamada de função `nome(x)` não corresponder a nenhuma função existente, mas `nome` for o nome de um tipo de dado, e `pg_cast` mostrar uma conversão binariamente compatível para o tipo de `x`, então a chamada será construída como uma conversão explícita. Esta exceção é feita para que as conversões compatíveis binariamente possam ser chamadas utilizando a sintaxe de função, embora não possuam nenhuma função.

## Exemplos

Para criar uma conversão do tipo `text` para o tipo `int4` utilizando a função `int4(text)`:

```
CREATE CAST (text AS int4) WITH FUNCTION int4(text);
```

(Esta conversão já está pré-definida no sistema).

## Compatibilidade

O comando `CREATE CAST` está em conformidade com o SQL:1999, exceto que o SQL:1999 não trata de tipos binariamente compatíveis nem de argumentos extra nas funções de implementação. A cláusula `AS IMPLICIT` também é uma extensão do PostgreSQL.

## Consulte também

*CREATE FUNCTION, CREATE TYPE, DROP CAST*

# CREATE CONSTRAINT TRIGGER

## Nome

CREATE CONSTRAINT TRIGGER — cria um gatilho de restrição

## Sinopse

```
CREATE CONSTRAINT TRIGGER nome
    AFTER eventos ON
    nome_da_tabela restrição atributos
    FOR EACH ROW EXECUTE PROCEDURE nome_da_função ( argumentos )
```

## Descrição

O comando CREATE CONSTRAINT TRIGGER é utilizado dentro do comando CREATE TABLE/ALTER TABLE e pelo pg\_dump, para criar gatilhos especiais para integridade referencial. Não se destina a uso geral.

## Parâmetros

*nome*

O nome do gatilho de restrição.

*eventos*

As categorias de evento para as quais este gatilho deve ser disparado.

*nome\_da\_tabela*

O nome (opcionalmente qualificado pelo esquema) da tabela onde ocorrem os eventos que disparam o gatilho.

*restrição*

A especificação da restrição.

*atributos*

Os atributos da restrição.

*nome\_da\_função*(*argumentos*)

A função a ser chamada como parte do processamento do gatilho.

# CREATE CONVERSION

## Nome

CREATE CONVERSION — cria uma conversão de codificação

## Sinopse

```
CREATE [DEFAULT] CONVERSION nome
    FOR codificação_de_origem TO codificação_de_destino FROM nome_da_função
```

## Descrição

O comando CREATE CONVERSION cria uma conversão de codificação. Os nomes das conversões podem ser utilizados na função convert para especificar uma determinada conversão de codificação. Além disso, as conversões marcadas como DEFAULT podem ser utilizadas para fazer a conversão automática de codificação entre o cliente e o servidor. Para esta finalidade devem ser criadas duas conversões: da codificação A para B e da codificação B para A.

Para poder criar uma conversão é necessário possuir o privilégio EXECUTE na função, e o privilégio CREATE no esquema de destino.

## Parâmetros

DEFAULT

A cláusula DEFAULT indica que esta é a conversão padrão para o caso particular destas codificações de origem e de destino. Em um esquema deve existir apenas uma codificação padrão para cada par de codificações.

*nome*

O nome da conversão. O nome da conversão pode ser qualificado pelo esquema. Caso não seja, a conversão é criada no esquema corrente. O nome da conversão deve ser único no esquema.

*codificação\_de\_origem*

O nome da codificação de origem.

*codificação\_de\_destino*

O nome da codificação de destino.

*nome\_da\_função*

A função utilizada para realizar a conversão. O nome da função pode ser qualificado pelo esquema. Caso não seja, a função é procurada no caminho.

A função deve possuir a seguinte assinatura:

```
funcao_de_conversao(
    integer, -- identificador da codificação de origem
    integer, -- identificador da codificação de destino
    cstring, -- cadeia de caracteres de origem (cadeia de caracteres C terminada por
nulo)
    cstring, -- cadeia de caracteres de destino (cadeia de caracteres C terminada por
nulo)
    integer -- comprimento da cadeia de caracteres de origem
) RETURNS void;
```

## Observações

Use o comando DROP CONVERSION para remover conversões definidas pelo usuário.

Os privilégios necessários para criar conversão podem ser alterados em uma versão futura.

## Exemplos

Para criar a conversão da codificação UNICODE para LATIN1 utilizando minha\_funcao:

```
CREATE CONVERSION minha_conversao FOR 'UNICODE' TO 'LATIN1' FROM minha_funcao;
```

## Compatibilidade

O comando `CREATE CONVERSION` é uma extensão do PostgreSQL. Não existe o comando `CREATE CONVERSION` no padrão SQL.

## Consulte também

*ALTER CONVERSION, CREATE FUNCTION, DROP CONVERSION*

# CREATE DATABASE

## Nome

`CREATE DATABASE` — cria um banco de dados

## Sinopse

```
CREATE DATABASE nome
    [ [ WITH ] [ OWNER [=] dono_do_banco_de_dados ]
      [ TEMPLATE [=] modelo ]
      [ ENCODING [=] codificação ]
      [ TABLESPACE [=] espaço_de_tabelas ] ]
```

## Descrição

O comando `CREATE DATABASE` cria um banco de dados no PostgreSQL.

Para poder criar um banco de dados é necessário ser um superusuário ou possuir o privilégio especial `CREATEDB`. Consulte o comando `CREATE USER`.

Normalmente, o criador se torna o dono do novo banco de dados. Os superusuários podem criar bancos de dados cujos donos são outros usuários utilizando a cláusula `OWNER` e, até mesmo, criar bancos de dados cujos donos são usuários sem nenhum privilégio especial. Usuários comuns com privilégio `CREATEDB` podem criar apenas bancos de dados cujos donos são eles mesmos.

Por padrão, o novo banco de dados é criado clonando o de banco de dados comum do sistema `template1`. Um modelo diferente pode ser especificado escrevendo `TEMPLATE modelo`. Em particular, escrevendo `TEMPLATE template0` pode ser criado um banco de dados básico contendo apenas os objetos comuns pré-definidos pela versão do PostgreSQL em uso. Esta forma é útil quando se deseja evitar a cópia de qualquer objeto da instalação local que possa ter sido adicionado ao `template1`.

## Parâmetros

*nome*

O nome do banco de dados a ser criado.

*dono\_do\_banco\_de\_dados*

O nome do usuário do banco de dados que será o dono do novo banco de dados, ou `DEFAULT` para usar o padrão (ou seja, o usuário que está executando o comando).

*modelo*

Nome do modelo a partir do qual o novo banco de dados será criado, ou `DEFAULT` para utilizar o modelo padrão (`template1`).

*codificação*

Codificação do conjunto de caracteres a ser utilizado no novo banco de dados. Deve ser especificada uma constante cadeia de caracteres (por exemplo, `'SQL_ASCII'`), ou o número inteiro da codificação, ou `DEFAULT` para utilizar a codificação padrão. Os conjuntos de caracteres suportados pelo PostgreSQL estão descritos na Seção 20.2.1.

*espaço\_de\_tabelas*

O nome do espaço de tabelas associado ao novo banco de dados, ou `DEFAULT` para utilizar o espaço de tabelas do banco de dados modelo. Este espaço de tabelas é o espaço de tabelas padrão para os objetos criados neste banco de dados. Consulte o comando `CREATE TABLESPACE` para obter informações adicionais.

Os parâmetros opcionais podem ser escritos em qualquer ordem, e não apenas na ordem mostrada acima.

## Observações

O comando `CREATE DATABASE` não pode ser executado dentro de um bloco de transação.

Erros contendo “could not initialize database directory” (não foi possível inicializar o diretório do banco de dados) estão normalmente relacionados com a falta de permissão no diretório de dados, disco cheio, ou outros problemas no sistema de arquivos.

Deve ser utilizado `DROP DATABASE` para remover o banco de dados.

O aplicativo *createdb*, fornecido por conveniência, é um programa em torno deste comando.

Embora seja possível copiar outro banco de dados em vez do `template1` especificando seu nome como modelo, não se pretende (ainda) que esta seja uma funcionalidade de “`COPY DATABASE`” de uso geral. Recomenda-se que os bancos de dados utilizados como modelo sejam tratados como se fossem somente para leitura. Consulte a Seção 18.3 para obter informações adicionais.

## Exemplos

Para criar um banco de dados:

```
CREATE DATABASE lusiadas;
```

Para criar o banco de dados `vendas` pertencendo ao usuário `usuvendas` com o espaço de tabelas padrão `espvendas`:

```
CREATE DATABASE vendas OWNER usuvendas TABLESPACE espvendas;
```

Para criar o banco de dados `musica` com suporte a conjunto de caracteres ISO-8859-1:

```
CREATE DATABASE musica ENCODING 'LATIN1';
```

## Compatibilidade

Não existe o comando `CREATE DATABASE` no padrão SQL. Os bancos de dados são equivalentes aos catálogos, cuja criação é definida pela implementação.

# CREATE DOMAIN

## Nome

CREATE DOMAIN — cria um domínio

## Sinopse

```
CREATE DOMAIN nome [AS] tipo_de_dado
    [ DEFAULT expressão ]
    [ restrição [ ... ] ]
```

onde *restrição* é:

```
[ CONSTRAINT nome_da_restrição ]
{ NOT NULL | NULL | CHECK (expressão) }
```

## Descrição

O comando CREATE DOMAIN cria um domínio de dados. O usuário que cria o domínio se torna o seu dono.<sup>1 2 3</sup>

Se o nome do esquema for fornecido (por exemplo, CREATE DOMAIN meu\_esquema.meu\_dominio ...), então o domínio é criado no esquema especificado, senão é criado no esquema corrente. O nome do domínio deve ser único entre os tipos e domínios existentes no esquema.

Domínios são úteis para abstrair campos comuns entre tabelas em um único local para manutenção. Por exemplo, uma coluna de endereço de correio eletrônico pode ser usada em várias tabelas, todas com as mesmas propriedades. Em vez de definir as restrições em cada tabela individualmente, deve ser definido e utilizado um domínio.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) do domínio a ser criado.

*tipo\_de\_dado*

O tipo de dado subjacente do domínio, podendo incluir especificadores de matrizes (arrays).

DEFAULT *expressão*

A cláusula DEFAULT especifica o valor padrão para as colunas com o tipo de dado do domínio. O valor é qualquer expressão sem variável (variable-free) (mas subconsultas não são permitidas). O tipo de dado da expressão padrão deve corresponder ao tipo de dado do domínio. Se o valor padrão não for especificado, então o valor nulo se torna o valor padrão.

A expressão padrão é usada em toda operação de inserção que não especifica valor para a coluna. Se for definido o valor padrão para uma determinada coluna, este substitui o valor padrão associado ao domínio. Por sua vez, o valor padrão para o domínio substitui o valor padrão associado ao tipo de dado subjacente.

CONSTRAINT *nome\_da\_restrição*

Um nome opcional para a restrição. Se não for especificado, o sistema gera um nome.

NOT NULL

Os valores deste domínio não podem ser nulos.

NULL

Os valores deste domínio podem ser nulos. Este é o padrão.

Esta cláusula tem somente a finalidade de manter a compatibilidade com os bancos de dados SQL fora do padrão. Seu uso é desaconselhado nos novos aplicativos.



CHECK (*expressão*)

A cláusula CHECK especifica as restrições de integridade, ou testes, que os valores do domínio devem satisfazer. Cada restrição deve ser uma expressão que produz um resultado booleano. Deve ser utilizado o nome VALUE para fazer referência ao valor sendo testado.

Atualmente as expressões CHECK não podem conter subconsultas, nem fazer referências a outras variáveis além de VALUE.

## Exemplos

Este exemplo cria o tipo de dado `us_postal_code` (código postal americano), e usa este tipo na definição da tabela. É utilizado um teste de expressão regular para verificar se o valor se parece com um código postal americano válido.

```
CREATE DOMAIN us_postal_code AS TEXT
CHECK (
    VALUE ~ '^d{5}$'
OR VALUE ~ '^d{5}-d{4}$'
);

CREATE TABLE us_snail_addy (
    address_id SERIAL NOT NULL PRIMARY KEY
, street1 TEXT NOT NULL
, street2 TEXT
, street3 TEXT
, city TEXT NOT NULL
, postal us_postal_code NOT NULL
);
```

## Compatibilidade

O comando `CREATE DOMAIN` está em conformidade com o padrão SQL.

## Consulte também

`ALTER DOMAIN`, `DROP DOMAIN`

## Notas

1. Um *domínio* é um objeto nomeado definido pelo usuário, que pode ser especificado como uma alternativa ao tipo de dado em certos locais onde o tipo de dado pode ser especificado. O domínio consiste do tipo de dado, possivelmente a opção `default`, e zero ou mais restrições (domínio).

A restrição de domínio se aplica a todas as colunas baseadas neste domínio, operando como uma restrição de tabela para cada coluna deste tipo.

As restrições de domínio se aplicam apenas às colunas baseadas no domínio associado.

A restrição de domínio é aplicada a todo valor resultante de uma operação de conversão (`cast`) para o domínio.

(ISO-ANSI Working Draft) Framework (SQL/Framework), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)

2. O SQL Server permite a especificação de tipos de dado. Os tipos de dado especificados pelo usuário consistem em um tipo de dado base do SQL Server, o comprimento do dado (se aplicável), a precisão do dado (se aplicável), e uma indicação da capacidade do dado em aceitar valores nulos. A criação dos tipos de dado definidos pelo usuário é feita através do procedimento armazenado `sp_addtype`. O domínio é o conjunto de todos os valores permitidos para a coluna. Inclui não somente o conceito de impor um tipo de dado, mas também os valores permitidos na coluna. Por exemplo, um domínio de cores poderia incluir tanto o tipo de dado, como `char(8)`, quanto as cadeias de caracteres permitidas na coluna, como Vermelho, Azul, Verde, Amarelo, Marrom, Preto, Branco, Ciano, Cinza e Prata. A obediência aos valores do domínio pode ser imposta através de restrições CHECK e gatilhos. SQL Server 2000 — SQL Server Books Online (N. do T.)
3. O DB2 também fornece suporte para tipos de dado definidos pelo usuário, que são classificados em três categorias: tipos distintos definidos pelo usuário (UDT), que permitem a criação de um novo tipo de dado, possuindo semântica própria, baseado em um tipo de dado existente; tipos estruturados definidos pelo usuário, que permitem a criação de

uma estrutura contendo uma sequência de atributos nomeados, cada um destes com seu tipo de dado, sendo esta uma das extensões das funções objeto-relacionais do DB2; e tipo referência definido pelo usuário, um tipo companheiro do tipo estruturado definido pelo usuário, que de forma semelhante ao tipo distinto definido pelo usuário é um tipo escalar que compartilha uma representação com um dos tipos de dado nativos, podendo ser usado para fazer referência a linhas em outra tabela que utilizam um tipo estruturado definido pelo usuário. DB2® Universal Database V8 for Linux, UNIX, and Windows Database Administration Certification Guide, 5th Edition (<http://www.phptr.com/title/0130463612>), George Baklarz e Bill Wong, Series IBM Press, Prentice Hall Professional Technical Reference, 2003, pág. 192. (N. do T.)

# CREATE FUNCTION

## Nome

CREATE FUNCTION — cria uma função

## Sinopse

```
CREATE [ OR REPLACE ] FUNCTION nome ( [ [ nome_do_argumento ] tipo_do_argumento [, ...] ] )
    RETURNS tipo_retornado
    { LANGUAGE nome_da_linguagem
      | IMMUTABLE | STABLE | VOLATILE
      | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
      | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
      | AS 'definição'
      | AS 'arquivo_objeto', 'símbolo_de_vinculação'
    } ...
    [ WITH ( atributo [, ...] ) ]
```

## Descrição

O comando CREATE FUNCTION cria uma função. O comando CREATE OR REPLACE FUNCTION cria uma função, ou substitui uma função existente.

Se o nome do esquema for incluído então a função é criada no esquema especificado, senão é criada no esquema corrente. O nome da nova função não deve ser igual ao de outra função existente com argumentos do mesmo tipo, no mesmo esquema. Entretanto, funções com argumentos de tipos diferentes podem ter o mesmo nome, o que é chamado de *sobrecarga* (overload).

Para atualizar a definição de uma função existente deve ser usado o comando CREATE OR REPLACE FUNCTION. Não é possível mudar o nome ou os tipos dos argumentos da função desta maneira; se for tentado, será criada uma nova função distinta. O comando CREATE OR REPLACE FUNCTION também não permite mudar o tipo de dado retornado por uma função existente; para fazer isto a função deve ser removida e recriada.

Se a função for removida e recriada, a nova função não é mais a mesma entidade que era antes; será necessário remover as regras, visões, gatilhos, etc. que fazem referência à função antiga. Use o comando CREATE OR REPLACE FUNCTION para mudar a definição de uma função sem invalidar os objetos que fazem referência à função.

O usuário que cria a função se torna o seu dono.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) da função a ser criada.

*nome\_do\_argumento*

O nome de um argumento. Algumas linguagens (atualmente apenas PL/pgSQL) deixam usar o nome no corpo da função. Para as demais linguagens o nome do argumento é apenas uma documentação adicional.

*tipo\_do\_argumento*

Os tipos de dado dos argumentos da função (opcionalmente qualificados pelo esquema), caso existam. O tipo de dado do argumento pode ser um tipo base, composto, ou domínio, ou pode fazer referência ao tipo de uma coluna de tabela.

Dependendo da linguagem de implementação também pode ser permitido especificar “pseudotipos” como *cstring*. Pseudotipos indicam que o tipo verdadeiro do argumento não está completamente especificado, ou está fora do conjunto comum de tipos de dado SQL.

O tipo de dado da coluna é referenciado escrevendo *nome\_da\_tabela.nome\_da\_coluna%TYPE*; a utilização desta notação pode, algumas vezes, ajudar a tornar a função independente das mudanças ocorridas na definição da tabela.

*tipo\_retornado*

O tipo de dado retornado (opcionalmente qualificados pelo esquema), que pode ser um tipo base, tipo composto ou domínio, ou pode fazer referência ao tipo de uma coluna de tabela. Dependendo da linguagem de implementação também pode ser permitido especificar “pseudotipos”, como `cstring`.

O modificador `SETOF` indica que a função retorna um conjunto de itens, em vez de um único item.

O tipo de dado da coluna é referenciado escrevendo `nome_da_tabela.nome_da_coluna%TYPE`.

*nome\_da\_linguagem*

O nome da linguagem usada para implementar a função. Pode ser `SQL`, `C`, `internal`, ou o nome de uma linguagem procedural definida pelo usuário. Para manter a compatibilidade com as versões anteriores, o nome pode estar entre apóstrofes (`'`).

`IMMUTABLE`

`STABLE`

`VOLATILE`

Estes atributos informam ao sistema se é seguro substituir várias chamadas à função por uma única chamada, para otimização em tempo de execução. Pode ser especificado, no máximo, um destes três atributos. Se nenhum deles for especificado, o padrão é assumir `VOLATILE`.

O atributo `IMMUTABLE` indica que a função sempre retorna o mesmo resultado quando recebe os mesmos valores para os argumentos, ou seja, não faz consultas a bancos de dados, ou de alguma outra forma utiliza informações que não estão diretamente presentes na sua lista de argumentos. Se esta opção for utilizada, qualquer chamada à função com todos os argumentos constantes pode ser imediatamente substituída pelo valor da função.

O atributo `STABLE` indica que dentro de uma única varredura da tabela a função retorna, consistentemente, o mesmo resultado para os mesmos valores dos argumentos, mas que seu resultado pode mudar entre comandos `SQL`. Esta é a seleção apropriada para as funções cujos resultados dependem de consultas a bancos de dados, valores de parâmetros (como a zona horária corrente), etc. Deve ser observado, também, que a família de funções `current_timestamp` se qualifica como estável, uma vez que seus valores não mudam dentro de uma transação.

O atributo `VOLATILE` indica que o valor da função pode mudar mesmo dentro de uma única varredura da tabela e, portanto, nenhuma otimização pode ser feita. Poucas funções de banco de dados são voláteis neste sentido; alguns exemplos são `random()`, `currval()` e `timeofday()`. Deve ser observado que toda função que produz efeito colateral deve ser classificada como volátil, mesmo que seu resultado seja totalmente previsível, para evitar que as chamadas sejam otimizadas; um exemplo é `setval()`.

Para obter detalhes adicionais consulte a Seção 31.6.

`CALLED ON NULL INPUT`

`RETURNS NULL ON NULL INPUT`

`STRICT`

`CALLED ON NULL INPUT` (o padrão) indica que a função é chamada normalmente quando algum de seus argumentos é nulo. Portanto, é responsabilidade do autor da função verificar a presença de valores nulos se for necessário, e responder de forma apropriada.

`RETURNS NULL ON NULL INPUT` ou `STRICT` indicam que a função sempre retorna nulo quando qualquer um de seus argumentos for nulo. Se este parâmetro for especificado, a função não será executada quando houver argumento nulo; em vez disto será assumido um resultado nulo automaticamente.

`[EXTERNAL] SECURITY INVOKER`

`[EXTERNAL] SECURITY DEFINER`

`SECURITY INVOKER` indica que a função deve ser executada com os privilégios do usuário a chamou. Este é o padrão. `SECURITY DEFINER` especifica que a função deve ser executada com os privilégios do usuário que a criou.

A palavra chave `EXTERNAL` está presente para manter a conformidade com o `SQL`. Entretanto é opcional porque, diferentemente do `SQL`, esta funcionalidade não se aplica apenas às funções externas.

*definição*

A cadeia de caracteres contendo a definição da função; o significado depende da linguagem. Pode ser o nome de uma função interna, o caminho para um arquivo objeto, um comando SQL, ou um texto escrito em uma linguagem procedural.

*arquivo\_objeto, símbolo\_de\_vinculação*

Esta forma da cláusula AS é utilizada para funções escritas na linguagem C carregáveis dinamicamente, quando o nome da função no código fonte na linguagem C não tem o mesmo nome da função SQL. A cadeia de caracteres *arquivo\_objeto* é o nome do arquivo contendo o objeto carregável dinamicamente, e *símbolo\_de\_vinculação* é o símbolo de vinculação da função, ou seja, o nome da função no código fonte na linguagem C. Se o símbolo de vinculação for omitido, é assumido como tendo o mesmo nome da função SQL sendo definida.

*atributo*

A forma histórica de especificar informações opcionais sobre a função. Os seguintes atributos podem ser utilizados:

*isStrict*

Equivalente a STRICT ou RETURNS NULL ON NULL INPUT.

*isCachable*

*isCachable* é um equivalente obsoleto de IMMUTABLE; ainda é aceito por motivo de compatibilidade com versões anteriores.

Não há diferença entre letras minúsculas de maiúsculas nos nomes de atributos.

## Observações

Consulte a Seção 31.3 para obter informações adicionais sobre como escrever funções.

A sintaxe tipo SQL completa é permitida para os argumentos de entrada e o valor retornado. Entretanto, alguns detalhes da especificação do tipo (por exemplo, o campo precisão para o tipo numeric) são de responsabilidade da implementação da função subjacente, sendo engolidos em silêncio (ou seja, não são reconhecidos nem exigidos) pelo comando CREATE FUNCTION.

O PostgreSQL permite a *sobrecarga* de função, ou seja, o mesmo nome pode ser utilizado por várias funções diferentes, desde que possuam argumentos com tipos distintos. Entretanto, na linguagem C os nomes de todas as funções devem ser diferentes e, portanto, as funções na linguagem C sobrecarregadas devem possuir nomes diferentes (por exemplo, utilizando os tipos dos argumentos como parte do nome na linguagem C).

Quando chamadas repetidas ao comando CREATE FUNCTION fazem referência ao mesmo arquivo objeto, o arquivo só é carregado uma vez. Para descarregar e recarregar o arquivo (talvez durante o desenvolvimento), deve ser usado o comando LOAD.

Use o comando DROP FUNCTION para remover funções definidas pelo usuário.

Geralmente é útil usar o caractere cifrão (\$) (consulte Seção 4.1.2.2) para envolver a cadeia de caracteres que define a função, em vez de usar a sintaxe normal de envolver por apóstrofes. Sem envolver a definição da função pelo caractere cifrão, todo apóstrofo ou contrabarra na definição da função deve receber um escape duplicando os mesmos.

Para poder criar uma função o usuário deve possuir o privilégio USAGE na linguagem.

## Exemplos

Abaixo estão mostrados exemplos simples para ajudar a começar. Para obter informações adicionais e exemplos deve ser consultada a Seção 31.3.

**Somar dois números inteiros.** Esta função recebe como argumentos dois números inteiros, e retorna como resultado a soma dos dois números recebidos.

```
CREATE FUNCTION soma(integer, integer) RETURNS integer
AS 'select $1 + $2;'
LANGUAGE SQL
IMMUTABLE
```

```
RETURNS NULL ON NULL INPUT;
```

**Incrementar um inteiro.** Incrementar um inteiro, fazendo uso do nome do argumento, no PL/pgSQL:

```
CREATE OR REPLACE FUNCTION incremento(i integer) RETURNS integer AS $$
BEGIN
    RETURN i + 1;
END;
$$ LANGUAGE plpgsql;
```

**Somar dias a uma data.** A seguir está mostrada uma função sobrecarregada para somar dias a data. Pode ser utilizada com os tipos de dado `date`, `timestamp` e `timestamp with time zone`.<sup>1</sup>

```
CREATE OR REPLACE FUNCTION soma_dias(data date,dias integer)
RETURNS date AS '
DECLARE
    nova_data date;
BEGIN
    nova_data := data + dias;
    RETURN nova_data;
END;
' LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION soma_dias(data timestamp,dias integer)
RETURNS timestamp AS '
DECLARE
    nova_data timestamp;
    hora interval;
BEGIN
    hora := data - CAST(data AS DATE);
    nova_data := CAST(data AS DATE)+ dias;
    RETURN nova_data + hora;
END;
' LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION soma_dias(data timestamp with time zone, dias integer)
RETURNS timestamp with time zone AS '
DECLARE
    nova_data timestamp with time zone;
    hora interval;
BEGIN
    hora := data - CAST(data AS DATE);
    nova_data := CAST(data AS DATE)+ dias;
    RETURN nova_data + hora;
END;
' LANGUAGE plpgsql;
```

```
SELECT soma_dias(date '2004-07-16', 30);
```

```
 soma_dias
-----
2004-08-15
(1 linha)
```

```
SELECT soma_dias(timestamp '2004-07-16 15:00:00', 30);
```

```
 soma_dias
-----
2004-08-15 15:00:00
(1 linha)
```

```
SELECT current_timestamp, soma_dias(current_timestamp, 30);
```

timestampz		soma_dias
-----+-----		
2005-02-15 21:29:05.403436-02		2005-03-17 21:29:05.403436-03
(1 linha)		

## Compatibilidade

O comando `CREATE FUNCTION` está definido no SQL:1999 e posterior. A versão do PostgreSQL é semelhante mas não é totalmente compatível. Os atributos não são portáteis, nem as diferentes linguagens disponíveis o são.

## Consulte também

*ALTER FUNCTION, DROP FUNCTION, GRANT, LOAD, REVOKE, createlang*

## Notas

1. Exemplo escrito pelo tradutor, não fazendo parte do manual original.

# CREATE GROUP

## Nome

`CREATE GROUP` — cria um grupo de usuários

## Sinopse

```
CREATE GROUP nome [ [ WITH ] opção [ ... ] ]
```

onde *opção* pode ser:

```
    SYSID id_do_grupo  
| USER nome_do_usuario [ , ... ]
```

## Descrição

O comando `CREATE GROUP` cria um grupo de usuários. É necessário ser um superusuário do banco de dados para executar este comando.

Deve ser observado que tanto os usuários quanto os grupos são definidos no nível de agrupamento de bancos de dados e, portanto, são válidos em todos os bancos de dados do agrupamento.

Deve ser usado o comando `ALTER GROUP` para incluir ou excluir usuários no grupo, e o comando `DROP GROUP` para remover um grupo.

## Parâmetros

*nome*

O nome do grupo.

*id\_do\_grupo*

A cláusula `SYSID` pode ser utilizada para escolher o identificador de grupo do PostgreSQL para o novo grupo. Normalmente não é necessário, mas pode ser útil se for necessário recriar um grupo referenciado pelas permissões de algum objeto.

Se não for especificado, o valor do identificador de grupo mais alto atribuído mais um (com um mínimo de 100) será utilizado por padrão.

*nome\_do\_usuario*

A lista dos usuários a serem incluídos no grupo. Os usuários devem existir.

## Exemplos

Criar um grupo vazio:

```
CREATE GROUP colaboradores;
```

Criar um grupo com membros:

```
CREATE GROUP vendas WITH USER jonas, marcela;
```

## Compatibilidade

Não existe o comando `CREATE GROUP` no padrão SQL. O conceito de “papéis” (`roles`) é semelhante ao de grupos.

## Consulte também

`ALTER GROUP`, `DROP GROUP`



# CREATE INDEX

## Nome

CREATE INDEX — cria um índice

## Sinopse

```
CREATE [ UNIQUE ] INDEX nome_do_índice ON tabela [ USING método ]  
    ( { coluna | ( expressão ) } [ classe_de_operadores ] [, ...] )  
    [ TABLESPACE espaço_de_tabelas ]  
    [ WHERE predicado ]
```

## Descrição

O comando `CREATE INDEX` constrói o índice *nome\_do\_índice* na tabela especificada. Os índices são utilizados, principalmente, para melhorar o desempenho do banco de dados (embora a utilização não apropriada cause uma degradação de desempenho).

Os campos chave para o índice são especificados como nomes de coluna ou, também, como expressões escritas entre parênteses. Podem ser especificados vários campos, se o método de índice suportar índices multicolunas.

O campo de um índice pode ser uma expressão computada a partir dos valores de uma ou mais colunas da linha da tabela. Esta funcionalidade pode ser utilizada para obter acesso rápido aos dados baseado em alguma transformação dos dados básicos. Por exemplo, um índice computado como `upper(col)` permite a cláusula `WHERE upper(col) = 'JIM'` utilizar um índice.

O PostgreSQL fornece os métodos de índice B-tree, R-tree, hash e GiST. O método de índice B-tree é uma implementação das B-trees de alta concorrência de Lehman-Yao <sup>1</sup>. O método de índice R-tree implementa R-trees padrão utilizando o algoritmo de divisão quadrática (*quadratic split*) de Guttman <sup>2</sup>. O método de índice hash é uma implementação do hash linear de Litwin <sup>3 4</sup>. Os usuários também podem definir seus próprios métodos de índice, mas é muito complicado.

Quando a cláusula `WHERE` está presente, um *índice parcial* é criado. Um índice parcial é um índice contendo entradas apenas para uma parte da tabela, geralmente uma parte mais útil para indexar que o restante da tabela. Por exemplo, havendo uma tabela contendo tanto pedidos faturados quanto não faturados, onde os pedidos não faturados ocupam uma pequena parte da tabela, mas que é a parte mais usada, o desempenho pode ser melhorado criando um índice apenas para esta parte da tabela. Outra aplicação possível é utilizar a cláusula `WHERE` junto com `UNIQUE` para impor a unicidade de um subconjunto dos dados da tabela. Consulte a Seção 11.7 para obter informações adicionais.

A expressão utilizada na cláusula `WHERE` pode referenciar apenas as colunas da tabela subjacente, mas pode usar todas as colunas, e não apenas as que estão sendo indexadas. Atualmente não são permitidas subconsultas e expressões de agregação na cláusula `WHERE`. As mesmas restrições se aplicam aos campos do índice que são expressões.

Todas as funções e operadores utilizados na definição do índice devem ser “imutáveis” (*immutable*), ou seja, seus resultados devem depender somente de seus argumentos, e nunca de uma influência externa (como o conteúdo de outra tabela ou a hora atual). Esta restrição garante que o comportamento do índice é bem definido. Para utilizar uma função definida pelo usuário na expressão do índice ou na cláusula `WHERE`, a função deve ser marcada como `IMMUTABLE` na sua criação.

## Parâmetros

**UNIQUE**

Faz o sistema verificar valores duplicados na tabela quando o índice é criado, se existirem dados, e toda vez que forem adicionados dados. A tentativa de inserir ou de atualizar dados que resultem em uma entrada duplicada gera um erro.

*nome\_do\_indice*

O nome do índice a ser criado. O nome do esquema não pode ser incluído aqui; o índice é sempre criado no mesmo esquema da tabela que este pertence.

*tabela*

O nome (opcionalmente qualificado pelo esquema) da tabela a ser indexada.

*método*

O nome do método a ser utilizado pelo índice. Pode ser escolhido entre `btree`, `hash`, `rtree` e `gist`. O método padrão é `btree`.

*coluna*

O nome de uma coluna da tabela.

*expressão*

Uma expressão baseada em uma ou mais colunas da tabela. A expressão geralmente deve ser escrita entre parênteses, conforme mostrado na sintaxe. Entretanto, os parênteses podem ser omitidos se a expressão tiver a forma de uma chamada de função.

*classe\_de\_operadores*

O nome de uma classe de operadores. Veja os detalhes abaixo.

*espaço\_de\_tabelas*

O espaço de tabelas onde o índice será criado. Se não for especificado, será utilizado o `default_tablespace`, ou o espaço de tabelas padrão do banco de dados se `default_tablespace` for uma cadeia de caracteres vazia.

*predicado*

A expressão de restrição para o índice parcial.

## Observações

Consulte o Capítulo 11 para obter informações sobre quando os índices podem ser utilizados, quando não são utilizados, e em quais situações particulares podem ser úteis.

Atualmente somente os métodos de índice `B-tree` e `Gist` suportam índices com mais de uma coluna. Por padrão podem ser especificadas até 32 colunas (este limite pode ser alterado na geração do PostgreSQL). Também atualmente somente `B-tree` suporta índices únicos.

Uma *classe de operadores* pode ser especificada para cada coluna de um índice. A classe de operadores identifica os operadores a serem utilizados pelo índice para esta coluna. Por exemplo, um índice `B-tree` sobre inteiros de quatro bytes usaria a classe `int4_ops`; esta classe de operadores inclui funções de comparação para inteiros de quatro bytes. Na prática, a classe de operadores padrão para o tipo de dado da coluna é normalmente suficiente. O ponto principal em haver classes de operadores é que, para alguns tipos de dado, pode haver mais de uma ordenação que faça sentido. Por exemplo, pode-se desejar classificar o tipo de dado do número complexo tanto pelo valor absoluto quanto pela parte real, o que pode ser feito definindo duas classes de operadores para o tipo de dado e, então, selecionando a classe apropriada na construção do índice. Mais informações sobre classes de operadores estão na Seção 11.6 e na Seção 31.14.

Deve ser utilizado o comando `DROP INDEX` para remover um índice.

Por padrão não é utilizado índice para a cláusula `IS NULL`. Nesta situação a melhor forma para utilizar índice é criar um índice parcial usando o predicado `IS NULL`.

## Exemplos

Para criar um índice `B-tree` para a coluna `titulo` na tabela `filmes`:

```
CREATE UNIQUE INDEX unq_titulo ON filmes (titulo);
```

Para criar um índice para a coluna `codigo` da tabela `filmes` e fazer o índice residir no espaço de tabelas `espaco_indices`:

```
CREATE INDEX idx_codigo ON filmes(codigo) TABLESPACE espaco_indices;
```

## Compatibilidade

O comando `CREATE INDEX` é uma extensão do PostgreSQL à linguagem. O padrão SQL não trata de índices.

## Consulte também

*ALTER INDEX, DROP INDEX*

## Notas

1. Lehman, Yao 81 - Philip L. Lehman , s. Bing Yao, Efficient locking for concurrent operations on B-trees, ACM Transactions on Database Systems (TODS), v.6 n.4, p.650-670, Dec. 1981 (N. do T.)
2. Antonin Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD Conference 1984. (N. do T.)
3. Litwin, W. Linear hashing: A new tool for file and table addressing. In Proceedings of the 6th Conference on Very Large Databases, (New York, 1980}, 212-223. (N. do T.)
4. Witold Litwin: Linear Hashing: A new Tool for File and Table Addressing (<http://swig.stanford.edu/pub/summaries/database/linhash.html>) - Summary by: Steve Gribble and Armando Fox. (N. do T.)

# CREATE LANGUAGE

## Nome

CREATE LANGUAGE — cria uma linguagem procedural

## Sinopse

```
CREATE [ TRUSTED ] [ PROCEDURAL ] LANGUAGE nome
    HANDLER tratador_de_chamadas [ VALIDATOR função_de_validação ]
```

## Descrição

Utilizando o comando `CREATE LANGUAGE`, um usuário do PostgreSQL pode registrar uma nova linguagem procedural em um banco de dados do PostgreSQL. Depois, podem ser definidas funções e procedimentos de gatilhos nesta nova linguagem. O usuário deve possuir o privilégio de superusuário do PostgreSQL para poder registrar uma nova linguagem.

O comando `CREATE LANGUAGE` associa o nome da linguagem ao tratador de chamadas (`call handler`) responsável pela execução das funções escritas nesta linguagem. Consulte a Seção 31.3 para obter informações adicionais sobre tratadores de chamada de linguagens.

Deve ser observado que as linguagens procedurais são locais a cada banco de dados. Para tornar, por padrão, uma linguagem disponível para todos os bancos de dados, esta deve ser instalada no banco de dados `template1`.

## Parâmetros

**TRUSTED**

`TRUSTED` especifica que o tratador de chamadas para a linguagem é seguro, ou seja, não oferece a um usuário sem privilégios qualquer funcionalidade para transpor as restrições de acesso. Se esta palavra chave for omitida ao registrar a linguagem, somente os usuários do PostgreSQL com privilégio de superusuário vão poder usar esta linguagem para criar novas funções.

**PROCEDURAL**

Apenas informativo.

*nome*

O nome da nova linguagem procedural. Não existe distinção entre letras minúsculas e maiúsculas no nome da linguagem. O nome deve ser único entre todas as linguagens do banco de dados.

Por compatibilidade com as versões anteriores, o nome pode ser escrito entre apóstrofes (`'`).

**HANDLER** *tratador\_de\_chamadas*

O *tratador\_de\_chamadas* é o nome de uma função, previamente registrada, que será chamada para executar as funções escritas nesta linguagem procedural. O tratador de chamadas para a linguagem procedural deve ser escrito em uma linguagem compilada como a linguagem C, com a convenção de chamadas versão 1, registrada no PostgreSQL como uma função que não recebe nenhum argumento e retorna o tipo `language_handler`, que é um tipo usado apenas para identificar a função como tratadora de chamadas.

**VALIDATOR** *função\_de\_validação*

A *função\_de\_validação* é nome de uma função, registrada previamente, que será chamada quando for criada uma nova função nesta linguagem para validar esta nova função. Se nenhuma função validadora for especificada, então a nova função não será verificada ao ser criada. A função validadora deve receber um argumento do tipo `oid`, que é o OID (identificador do objeto) da função a ser criada, e normalmente retorna `void`.

Tipicamente a função validadora inspeciona o corpo da função para verificar se a sintaxe está correta, mas também pode verificar outras propriedades da função como, por exemplo, a linguagem não poder tratar argumentos de determinados tipos. Para sinalizar erro a função validadora deve usar a função `ereport()`. O valor retornado pela função é ignorado.

## Observações

Normalmente, este comando não deve ser executado diretamente pelos usuários. Para as linguagens procedurais fornecidas na distribuição do PostgreSQL deve ser utilizado o aplicativo `createlang`, porque este aplicativo também instala o tratador de chamadas correto; o aplicativo `createlang` chama o comando `CREATE LANGUAGE` internamente.

Nas versões do PostgreSQL anteriores a 7.3, era necessário declarar as funções tratadoras como retornando o tipo `opaque`, em vez de `language_handler`. Para permitir a carga de cópias de segurança antigas, o comando `CREATE LANGUAGE` aceita as funções declaradas como retornando `opaque`, mas mostra uma mensagem e muda para `language_handler` o tipo retornado declarado pela função.

Use o comando `CREATE FUNCTION` para criar uma função.

Use o comando `DROP LANGUAGE`, ou melhor ainda, o aplicativo `droplang`, para excluir linguagens procedurais.

O catálogo do sistema `pg_language` (consulte a Seção 41.18) registra informações sobre as linguagens instaladas correntemente. O aplicativo `createlang` possui uma opção para listar as linguagens instaladas.

Para poder utilizar uma linguagem procedural, deve ser concedido o privilégio `USAGE` para o usuário. O aplicativo `createlang` concede, automaticamente, permissão para todos se a linguagem for sabidamente `trusted`.

## Exemplos

Os dois comandos mostrados abaixo, executados em seqüência, registram uma nova linguagem procedural e o tratador de chamadas associado:

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

## Compatibilidade

O comando `CREATE LANGUAGE` é uma extensão do PostgreSQL.

## Consulte também

*ALTER LANGUAGE*, *CREATE FUNCTION*, *DROP LANGUAGE*, *GRANT*, *REVOKE*, *createlang*, *droplang*

# CREATE OPERATOR

## Nome

CREATE OPERATOR — cria um operador

## Sinopse

```
CREATE OPERATOR nome (  
    PROCEDURE = nome_da_função  
    [, LEFTARG = tipo_à_esquerda ] [, RIGHTARG = tipo_à_direita ]  
    [, COMMUTATOR = comutador_do_operador ] [, NEGATOR = negador_do_operador ]  
    [, RESTRICT = função_de_seletividade_da_restrição ] [, JOIN = função_de_seletividade_da_junção ]  
    [, HASHES ] [, MERGES ]  
    [, SORT1 = operador_de_ordenação_à_esquerda ] [, SORT2 = operador_de_ordenação_à_direita ]  
    [, LTCMP = operador_menor_que ] [, GTCMP = operador_maior_que ]  
)
```

## Descrição

O comando `CREATE OPERATOR` define o novo operador, *nome*. O usuário que define o operador se torna seu dono. Se for fornecido o nome do esquema, então o operador será criado no esquema especificado, senão será criado no esquema corrente.

O nome do operador é uma sequência com até NAMEDATALEN-1 (63, por padrão) caracteres da seguinte lista:

+ - \* / < > = ~ ! @ # % ^ & | ` ?

Existem algumas restrições na escolha do nome:

- As seqüências `--` e `/*` não podem ocorrer em nenhum lugar no nome do operador, uma vez que são consideradas início de comentário.
- Um nome de operador multicaractere não pode terminar por `+` ou por `-`, a não ser que o nome também contenha pelo menos um destes caracteres:

~ ! @ # % ^ & | ` ?

Por exemplo, `@-` é um nome de operador permitido, mas `*-` não é. Esta restrição permite ao PostgreSQL analisar comandos em conformidade com o SQL sem requerer espaços entre os elementos (tokens).

O operador `!=` é mapeado para `<>` na entrada e, portanto, estes dois nomes são sempre equivalentes.

Com relação a `LEFTARG` e `RIGHTARG`, pelo menos um dos dois deve ser definido; para operadores binários os dois devem ser definidos. Para operadores unários direito somente `LEFTARG` deve ser definido, enquanto que para operadores unários esquerdo somente `RIGHTARG` deve ser definido.

O procedimento *nome\_da\_função* deve ser previamente definido utilizando o comando `CREATE FUNCTION`, e deve estar definido para aceitar o número correto de argumentos (um ou dois) dos tipos indicados.

As outras cláusulas são cláusulas de otimização de operador opcionais. Seus significados estão descritos na Seção 31.12.

## Parâmetros

*nome*

O nome do operador a ser definido. Veja acima os caracteres permitidos. O nome pode ser qualificado pelo esquema como, por exemplo, `CREATE OPERATOR meu_esquema.+ (...)`; caso não seja, o operador será criado no esquema corrente. Dois operadores no mesmo esquema podem possuir o mesmo nome se operarem sobre tipos de dado diferentes. Isto se chama *sobrecarga* (overload).

*nome\_da\_função*

A função utilizada para implementar este operador.

*tipo\_à\_esquerda*

O tipo de dado do operando à esquerda do operador, se houver. Esta opção deve ser omitida em operadores unário esquerdo.

*tipo\_à\_direita*

O tipo de dado do operando à direita do operador, se houver. Esta opção deve ser omitida em operadores unário direito.

*comutador\_do\_operador*

O comutador deste operador.

*negador\_do\_operador*

O negador deste operador.

*função\_de\_seletividade\_da\_restrição*

A função que estima a seletividade da restrição para este operador.

*função\_de\_seletividade\_da\_junção*

A função que estima a seletividade da junção para este operador.

HASHES

Indica que este operador pode suportar uma junção por hash.

MERGES

Indica que este operador pode suportar uma junção por mesclagem.

*operador\_de\_ordenação\_à\_esquerda*

Se este operador puder suportar uma junção por mesclagem, o operador menor-que que classifica o tipo de dado à esquerda deste operador.

*operador\_de\_ordenação\_à\_direita*

Se este operador puder suportar uma junção por mesclagem, o operador menor-que que classifica o tipo de dado à direita deste operador.

*operador\_menor\_que*

Se este operador puder suportar uma junção por mesclagem, o operador menor-que que compara os tipos de dado de entrada deste operador.

*operador\_maior\_que*

Se este operador puder suportar uma junção por mesclagem, o operador maior-que que compara os tipos de dado de entrada deste operador.

Para usar um nome de operador qualificado pelo esquema em *comutador\_do\_operador*, ou nos demais argumentos opcionais, deve ser utilizada a sintaxe de `OPERATOR ( )` como, por exemplo,

```
COMMUTATOR = OPERATOR (meu_esquema.==) ,
```

## Observações

Consulte a Seção 31.12 para obter informações adicionais.

Utilize o comando *DROP OPERATOR* para remover do banco de dados operadores definidos pelo usuário. Utilize o comando *ALTER OPERATOR* para modificar os operadores no banco de dados.

## Exemplos

O comando abaixo define um novo operador, *area-equality* (igualdade de área), para o tipo de dado *box*:

```

CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    PROCEDURE = area_equal_procedure,
    COMMUTATOR = ===,
    NEGATOR = !=,
    RESTRICT = area_restriction_procedure,
    JOIN = area_join_procedure,
    HASHES,
    SORT1 = <<<,
    SORT2 = <<<
    -- Uma vez que os operadores de classificação foram fornecidos, MERGES está envolvido.
    -- LTCMP e GTCMP são assumidos como sendo < e >, respectivamente
);

```

## Compatibilidade

O comando `CREATE OPERATOR` é uma extensão do PostgreSQL. O padrão SQL não trata de operadores definidos pelo usuário.

## Consulte também

*ALTER OPERATOR, CREATE OPERATOR CLASS, DROP OPERATOR*



# CREATE OPERATOR CLASS

## Nome

CREATE OPERATOR CLASS — cria uma classe de operadores

## Sinopse

```
CREATE OPERATOR CLASS nome [ DEFAULT ] FOR TYPE tipo_de_dado USING método_de_índice AS
{
  OPERATOR número_da_estratégia nome_do_operador [ ( tipo_do_operador, tipo_do_operador
) ] [ RECHECK ]
  | FUNCTION número_de_suporte nome_da_função ( tipo_do_argumento [, ...] )
  | STORAGE tipo_armazenado
} [, ... ]
```

## Descrição

O comando CREATE OPERATOR CLASS cria uma classe de operadores. Uma classe de operadores define como um determinado tipo de dado pode ser usado em um índice. A classe de operadores especifica que certos operadores vão preencher determinados papéis, ou “estratégias”, para este tipo de dado e este método de índice. A classe de operadores também especifica os procedimentos de suporte a serem usados pelo método do índice quando a classe de operadores é selecionada para uma coluna do índice. Todos os operadores e funções usados por uma classe de operadores devem ser definidos antes da classe de operadores ser criada.

Se o nome do esquema for fornecido, então a classe de operadores é criada no esquema especificado, senão é criada no esquema corrente. Duas classes de operadores no mesmo esquema podem ter o mesmo nome somente se forem para métodos de índice diferentes.

O usuário que cria a classe de operadores se torna seu dono. Atualmente o usuário criador deve ser um superusuário; esta restrição é feita porque uma definição de classe de operadores errada pode confundir, ou mesmo derrubar, o servidor.

Atualmente o comando CREATE OPERATOR CLASS não verifica se a definição da classe de operadores inclui todos os operadores e funções requeridos pelo método de índice. É responsabilidade do usuário definir uma classe de operadores válida.

Consulte a Seção 31.14 para obter informações adicionais.

## Parâmetros

*nome*

O nome da classe de operadores a ser criada. O nome pode ser qualificado pelo esquema.

DEFAULT

Se estiver presente, a classe de operadores se tornará a classe de operadores padrão para seu tipo de dado. No máximo uma classe de operadores pode ser a classe padrão para um determinado tipo de dado e método de índice.

*tipo\_de\_dado*

O tipo de dado de coluna que esta classe de operadores se destina.

*método\_de\_índice*

O nome do método de índice que esta classe de operadores se destina.

*número\_da\_estratégia*

O número da estratégia do método de índice para um operador associado com a classe de operadores.

*nome\_do\_operador*

O nome (opcionalmente qualificado pelo esquema) de um operador associado com a classe de operadores.

*tipo\_do\_operador*

Os tipos de dado dos operandos de um operador, ou NONE indicando um operador unário-esquerdo ou unário-direito. Os tipos de dado dos operandos podem ser omitidos no caso usual, onde são iguais ao tipo de dado da classe de operadores.

*RECHECK*

Se estiver presente, o índice para este operador é “lossy”<sup>1</sup> (com perdas) e, portanto, as linhas trazidas usando o índice devem ser verificadas outra vez para ver se realmente satisfazem a cláusula de qualificação envolvendo este operador.

*número\_de\_suporte*

O número do procedimento de suporte do método de índice para a função associada com a classe de operadores.

*nome\_da\_função*

O nome (opcionalmente qualificado pelo esquema) da função que é o procedimento de suporte do método de índice para a classe de operadores.

*tipos\_dos\_argumentos*

Os tipos de dado dos parâmetros da função.

*tipo\_armazenado*

O tipo de dado realmente armazenado no índice. Geralmente é o mesmo tipo de dado da coluna, mas alguns métodos de índice (somente GIST no momento) permitem que seja diferente. A cláusula STORAGE deve ser omitida, a menos que o método de índice permita o uso de um tipo diferente.

As cláusulas OPERATOR, FUNCTION e STORAGE podem ser escritas em qualquer ordem.

## Observações

Os operadores não devem ser definidos por funções SQL. É possível que a função SQL seja incorporada (inlined<sup>2</sup>) ao comando que faz a chamada, não permitindo que o otimizador reconheça que o comando corresponde a um índice.

## Exemplos

O exemplo mostrado abaixo define uma classe de operadores de índice GiST para o tipo de dado \_int4 (matriz de int4). Consulte contrib/intarray/ para ver o exemplo completo.

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR          3      &&,
        OPERATOR          6      =          RECHECK,
        OPERATOR          7      @,
        OPERATOR          8      ~,
        OPERATOR          20     @@ (_int4, query_int),
        FUNCTION           1      g_int_consistent (internal, _int4, int4),
        FUNCTION           2      g_int_union (bytea, internal),
        FUNCTION           3      g_int_compress (internal),
        FUNCTION           4      g_int_decompress (internal),
        FUNCTION           5      g_int_penalty (internal, internal, internal),
        FUNCTION           6      g_int_picksplit (internal, internal),
        FUNCTION           7      g_int_same (_int4, _int4, internal);
```

## Compatibilidade

O comando CREATE OPERATOR CLASS é uma extensão do PostgreSQL. Não existe o comando CREATE OPERATOR CLASS no padrão SQL.

## Consulte também

ALTER OPERATOR CLASS, DROP OPERATOR CLASS

## Notas

1. `lossy` — Termo que descreve um algoritmo de compressão que na verdade reduz a quantidade de informações nos dados, em vez de reduzir apenas o número de bits usados para representar esta informação. A informação perdida normalmente é removida porque é subjetivamente menos importante à qualidade dos dados (geralmente uma imagem ou som), ou porque pode ser recuperada razoavelmente por interpolação dos dados remanescentes. MPEG e JPEG são exemplos de técnicas de compressão com perdas. FOLDOC - Free On-Line Dictionary of Computing (<http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=lossy>) (N. do T.)
2. `inline` — Substituir a chamada à função por uma instância do corpo da função. FOLDOC - Free On-Line Dictionary of Computing (<http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=inline>) (N. do T.)

# CREATE RULE

## Nome

CREATE RULE — cria uma regra de reescrita

## Sinopse

```
CREATE [ OR REPLACE ] RULE nome AS ON evento
    TO tabela [ WHERE condição ]
    DO [ ALSO | INSTEAD ] { NOTHING | comando | ( comando ; comando ... ) }
```

## Descrição

O comando CREATE RULE cria uma regra aplicada à tabela ou visão especificada. O comando CREATE OR REPLACE RULE cria uma regra, ou substitui uma regra existente com mesmo nome, na tabela.

O sistema de regras do PostgreSQL permite definir uma ação alternativa a ser realizada nas inclusões, atualizações ou exclusões em tabelas do banco de dados. Sem entrar em detalhes, uma regra faz com que comandos adicionais sejam executados quando um determinado comando é executado em uma determinada tabela. Diferentemente, a regra INSTEAD pode substituir um determinado comando por outro, ou mesmo fazer com que o comando não seja executado. As regras também são utilizadas para implementar as visões das tabelas. É importante perceber que a regra é, na realidade, um mecanismo de transformação de comando, ou uma macro de comando. A transformação acontece antes do início da execução do comando. Se, na verdade, for desejada uma operação que dispare de forma independente para cada linha física, provavelmente o que se deseja é um gatilho, e não uma regra. Mais informações sobre o sistema de regras podem ser obtidas no Capítulo 33.

Atualmente, as regras ON SELECT devem ser regras INSTEAD incondicionais, e devem possuir ações consistindo de um único comando SELECT. Portanto, uma regra ON SELECT tem por efeito transformar a tabela em uma visão, cujo conteúdo visível são as linhas retornadas pelo comando SELECT da regra, em vez do que está armazenado na tabela (se houver alguma coisa). É considerado um estilo melhor usar o comando CREATE VIEW do que criar uma tabela real e definir uma regra ON SELECT para a mesma.

É possível criar a ilusão de uma visão atualizável definindo regras ON INSERT, ON UPDATE e ON DELETE, ou qualquer subconjunto destas que seja suficiente para as finalidades desejadas, para substituir as ações de atualização na visão por atualizações apropriadas em outras tabelas.

Existe algo a ser lembrado quando se tenta utilizar regras condicionais para atualização de visões: é *obrigatório* haver uma regra incondicional INSTEAD para cada ação que se deseja permitir na visão. Se a regra for condicional, ou não for INSTEAD, então o sistema continuará a rejeitar as tentativas de realizar a ação de atualização, porque acha que poderá acabar tentando realizar a ação sobre a tabela fictícia da visão em alguns casos. Se for desejado tratar todos os casos úteis por meio de regras condicionais, deve ser adicionada uma regra incondicional DO INSTEAD NOTHING para garantir que o sistema sabe que nunca será chamado para atualizar a tabela fictícia. Em seguida devem ser criadas as regras condicionais não-INSTEAD; nos casos onde se aplicam, se adicionam à ação padrão INSTEAD NOTHING.

## Parâmetros

*nome*

O nome da regra a ser criada, devendo ser distinto do nome de qualquer outra regra para a mesma tabela. Havendo várias regras para a mesma tabela e mesmo tipo de evento, estas regras são aplicadas na ordem alfabética dos nomes.

*evento*

Evento é um entre SELECT, INSERT, UPDATE e DELETE.

*tabela*

O nome (opcionalmente qualificado pelo esquema) da tabela ou da visão à qual a regra se aplica.

*condição*

Qualquer expressão condicional SQL (retornando `boolean`). A expressão condicional não pode fazer referência a nenhuma tabela, exceto `NEW` e `OLD`, e não pode conter funções de agregação.

*INSTEAD*

`INSTEAD` indica que os comandos devem ser executados *em vez dos* (`instead of`) comandos originais.

*ALSO*

`ALSO` indica que os comandos devem ser executados *adicionalmente* aos comandos originais.

Se não for especificado nem `ALSO` nem `INSTEAD`, `ALSO` é o padrão.

*comando*

O comando ou comandos que compõem a ação da regra. Os comandos válidos são `SELECT`, `INSERT`, `UPDATE`, `DELETE` e `NOTIFY`.

Dentro da *condição* e do *comando*, os nomes especiais de tabela `NEW` e `OLD` podem ser usados para fazer referência aos valores na tabela referenciada. O `NEW` é válido nas regras `ON INSERT` e `ON UPDATE`, para fazer referência à nova linha sendo inserida ou atualizada. O `OLD` é válido nas regras `ON UPDATE` e `ON DELETE`, para fazer referência à linha existente sendo atualizada ou excluída.

## Observações

É necessário possuir o privilégio `RULE` na tabela para poder definir uma regra para a mesma.

É muito importante tomar cuidado para evitar regras circulares. Por exemplo, embora as duas definições de regra abaixo sejam aceitas pelo PostgreSQL, o comando `SELECT` faz com que o PostgreSQL relate um erro, porque a consulta vai circular muitas vezes:

```
CREATE RULE "_RETURN" AS
  ON SELECT TO t1
  DO INSTEAD
    SELECT * FROM t2;
```

```
CREATE RULE "_RETURN" AS
  ON SELECT TO t2
  DO INSTEAD
    SELECT * FROM t1;
```

```
SELECT * FROM t1;
```

Atualmente, se a ação da regra contiver um comando `NOTIFY`, este comando `NOTIFY` será executado incondicionalmente, ou seja, o `NOTIFY` será emitido mesmo não havendo nenhuma linha onde a regra se aplique. Por exemplo, em

```
CREATE RULE me_notifique AS ON UPDATE TO minha_tabela DO ALSO NOTIFY minha_tabela;
```

```
UPDATE minha_tabela SET nome = 'foo' WHERE id = 42;
```

um evento `NOTIFY` será enviado durante o `UPDATE`, haja ou não alguma linha que corresponda à condição `id = 42`. Esta é uma restrição da implementação que deverá estar corrigida em versões futuras.

## Compatibilidade

O comando `CREATE RULE` é uma extensão do PostgreSQL à linguagem, assim como todo o sistema de reescrita de comandos.

# CREATE SCHEMA

## Nome

CREATE SCHEMA — cria um esquema

## Sinopse

```
CREATE SCHEMA nome_do_esquema [ AUTHORIZATION nome_do_usuario ] [ elemento_do_esquema [ ... ] ]  
CREATE SCHEMA AUTHORIZATION nome_do_usuario [ elemento_do_esquema [ ... ] ]
```

## Descrição

O comando CREATE SCHEMA cria um esquema no banco de dados corrente. O nome do esquema deve ser distinto do nome de todos os outros esquemas existentes no banco de dados corrente.

Um esquema é essencialmente um espaço de nomes: contém objetos nomeados (tabelas, tipos de dado, funções e operadores) cujos nomes podem ser iguais aos de outros objetos existentes em outros esquemas. Os objetos nomeados são acessados “qualificando” seus nomes usando o nome do esquema como prefixo, ou definindo um caminho de procura que inclua os esquemas desejados. Os objetos não qualificados são criados no esquema corrente (o primeiro do caminho de procura, que pode ser determinado pela função `current_schema()`).

Opcionalmente, o comando CREATE SCHEMA pode incluir subcomandos para criar objetos no novo esquema. Estes subcomandos são tratados, essencialmente, da mesma maneira como são tratados os comandos isolados executados após a criação do esquema, exceto que, se a cláusula AUTHORIZATION for usada, todos os objetos criados pertencerão a este usuário.

## Parâmetros

*nome\_do\_esquema*

O nome do esquema a ser criado. Se for omitido, o nome do usuário será usado como o nome do esquema. O nome não pode começar por `pg_`, porque estes nomes são reservados para os esquemas do sistema.

*nome\_do\_usuario*

O nome do usuário que será o dono do esquema. Se for omitido, será usado por padrão o usuário executando o comando. Somente os superusuários podem criar esquemas pertencentes a outros usuários.

*elemento\_do\_esquema*

Um comando SQL definindo um objeto a ser criado no esquema. Atualmente, somente as cláusulas CREATE TABLE, CREATE VIEW, CREATE INDEX, CREATE SEQUENCE, CREATE TRIGGER e GRANT são aceitas no comando CREATE SCHEMA. Os outros tipos de objeto podem ser criados por comandos separados, após o esquema ter sido criado.

## Observações

Para criar um esquema, o usuário deve possuir o privilégio CREATE no banco de dados corrente (É claro que os superusuários não são afetados por esta exigência).

## Exemplos

Criar um esquema:

```
CREATE SCHEMA meu_esquema;
```

Criar um esquema para o usuário antonio; o esquema também se chamará antonio:

```
CREATE SCHEMA AUTHORIZATION antonio;
```

Criar um esquema e criar uma tabela e uma visão nele:

```
CREATE SCHEMA hollywood
    CREATE TABLE filmes (titulo text, lancamento date, premios text[])
    CREATE VIEW premiados AS
        SELECT titulo, lancamento FROM filmes WHERE premios IS NOT NULL;
```

Deve ser observado que os subcomandos individuais não terminam por ponto-e-vírgula.

Abaixo está mostrada uma forma equivalente para se obter o mesmo resultado:

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.filmes (titulo text, lancamento date, premios text[])
CREATE VIEW hollywood.premiados AS
    SELECT titulo, lancamento FROM hollywood.filmes WHERE premios IS NOT NULL;
```

## Compatibilidade

O padrão SQL permite a cláusula `DEFAULT CHARACTER SET` no comando `CREATE SCHEMA`, bem como mais tipos de subcomandos que os aceitos atualmente pelo PostgreSQL.

O padrão SQL especifica que os subcomandos presentes em `CREATE SCHEMA` podem estar em qualquer ordem. A implementação atual do PostgreSQL não trata todos os casos de referência à frente nos subcomandos; alguma vezes pode ser necessário reordenar os subcomandos para evitar referências à frente.

De acordo com o padrão SQL, o dono do esquema sempre possui todos os objetos que este contém. O PostgreSQL permite que os esquemas contenham objetos pertencentes a outros usuários. Isto só acontece quando o dono do esquema concede o privilégio `CREATE` em seu esquema para algum outro usuário.

## Consulte também

*ALTER SCHEMA*, *DROP SCHEMA*

## Notas

1. Use o comando `SELECT current_schema();` (N. do T.)

# CREATE SEQUENCE

## Nome

CREATE SEQUENCE — cria um gerador de seqüência

## Sinopse

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE nome [ INCREMENT [ BY ] incremento ]  
    [ MINVALUE valor_mínimo | NO MINVALUE ] [ MAXVALUE valor_máximo | NO MAXVALUE ]  
    [ START [ WITH ] início ] [ CACHE cache ] [ [ NO ] CYCLE ]
```

## Descrição

O comando CREATE SEQUENCE cria um gerador de números seqüenciais, que envolve a criação e a inicialização de uma nova tabela especial com uma única linha chamada *nome*. O usuário que executa o comando se torna o dono do gerador.

Se um nome de esquema for fornecido, então a seqüência é criada no esquema especificado, senão é criada no esquema corrente. As seqüências temporárias são criadas em um esquema especial e, portanto, o nome do esquema não pode ser fornecido ao se criar uma seqüência temporária. O nome da seqüência deve ser distinto do nome de qualquer outra seqüência, tabela, índice ou visão no mesmo esquema.

Após a seqüência ser criada, podem ser utilizadas as funções nextval, currval e setval para operar na seqüência. Estas funções estão documentadas na Seção 9.12.

Embora não seja possível atualizar uma seqüência diretamente, pode ser feita uma consulta como

```
SELECT * FROM nome;
```

para examinar os parâmetros e o estado atual da seqüência. Em particular, o campo last\_value da seqüência mostra o último valor atribuído para qualquer sessão (É claro que este valor pode estar obsoleto no instante em que for exibido, se outras sessões estiverem chamando a função nextval).

## Parâmetros

TEMPORARY ou TEMP

Se for especificado, o objeto de seqüência é criado somente para esta sessão, sendo automaticamente removido ao término da sessão. Seqüências permanentes existentes não serão visíveis (na sessão) enquanto existirem seqüências temporárias com o mesmo nome, a não ser que sejam referenciadas por um nome qualificado pelo esquema.

*nome*

O nome (opcionalmente qualificado pelo esquema) da seqüência a ser criada.

*incremento*

A cláusula opcional INCREMENT BY *incremento* especifica o valor a ser adicionado ao valor corrente da seqüência para gerar o novo valor. Um valor positivo cria uma seqüência ascendente, enquanto um valor negativo cria uma seqüência descendente. O valor padrão é 1.

*valor\_mínimo*

NO MINVALUE

A cláusula opcional MINVALUE *valor\_mínimo* determina o valor mínimo que a seqüência pode gerar. Se esta cláusula não for fornecida, e se NO MINVALUE não for especificado, então são utilizados os valores padrão. Os valores padrão são 1 e  $-2^{63}-1$  para seqüências ascendentes e descendentes, respectivamente.

*valor\_máximo*

NO MAXVALUE

A cláusula opcional MAXVALUE *valor\_máximo* determina o valor máximo que a seqüência pode gerar. Se esta cláusula não for fornecida, e se NO MAXVALUE não for especificado, então são utilizados os valores padrão. Os valores padrão são  $2^{63}-1$  e -1 para seqüências ascendentes e descendentes, respectivamente.



*início*

A cláusula opcional `START WITH início` permite a sequência iniciar com qualquer valor. O valor inicial padrão é o *valor\_mínimo* para seqüências ascendentes, e o *valor\_máximo* para seqüências descendentes.

*cache*

A cláusula opcional `CACHE cache` especifica quantos números da sequência são previamente reservados e armazenados em memória para acesso mais rápido. O valor mínimo é 1 (somente um valor é gerado de cada vez, ou seja, sem cache), e este também é o valor padrão.

## CYCLE

## NO CYCLE

A opção `CYCLE` permite uma sequência reiniciar quando atingir o *valor\_máximo* ou o *valor\_mínimo*, respectivamente. Se o limite for atingido, o próximo número gerado será o *valor\_mínimo* ou o *valor\_máximo*, respectivamente.

Se `NO CYCLE` for especificado, toda chamada a `nextval` após a sequência ter atingido seu valor máximo retorna um erro. Se nem `CYCLE` nem `NO CYCLE` for especificado, `NO CYCLE` é o padrão.

## Observações

Use o comando `DROP SEQUENCE` para remover uma sequência.

As seqüências são baseadas na aritmética do tipo `bigint` e, portanto, a faixa de valores não pode exceder a faixa de um número inteiro de 8 bytes (-9223372036854775808 a 9223372036854775807). Em algumas plataformas mais antigas, pode não haver suporte do compilador para números inteiros de 8 bytes e, neste caso, as seqüências utilizam a aritmética do tipo `integer` regular (faixa de valores de -2147483648 a +2147483647).

Se for utilizada uma definição de *cache* maior que um, para um objeto de sequência que será usado concorrentemente por várias sessões, podem ser obtidos resultados não esperados. Cada sessão reserva e armazena valores sucessivos da sequência durante um acesso ao objeto de sequência, e aumenta o valor de `last_value` do objeto de sequência da forma apropriada. Então, as próximas *cache*-1 utilizações de `nextval` nesta sessão simplesmente retornam os valores reservados sem acessar o objeto de sequência. Portanto, todos os números alocados, mas não utilizados pela sessão, são perdidos quando a sessão termina, produzindo “buracos” na sequência.

Além disso, embora se garanta que as várias sessões reservam valores distintos da sequência, os valores podem ser gerados fora de sequência quando são levadas em consideração todas as sessões. Por exemplo, definindo-se *cache* igual a 10, a sessão A pode reservar os valores 1..10 e retornar `nextval`=1, enquanto a sessão B pode reservar os valores 11..20 e retornar `nextval`=11 antes da sessão A ter gerado `nextval`=2. Portanto, com uma definição de *cache* igual a um é seguro assumir que os valores de `nextval` são gerados seqüencialmente; com uma definição de *cache* maior do que um apenas pode-se assumir que os valores de `nextval` são todos distintos, mas não que sejam gerados de forma inteiramente seqüencial. Também, o valor `last_value` reflete o último valor reservado por qualquer sessão, tenha ou não sido retornado por `nextval`.

Outra consideração a ser feita é que a execução de `setval` neste tipo de sequência não será percebida pelas outras sessões enquanto estas não utilizarem todos os valores reservados guardados no cache.

## Exemplos

Criar uma sequência ascendente chamada `serial`, começando por 101:

```
CREATE SEQUENCE serial START 101;
```

Selecionar o próximo valor desta sequência:

```
SELECT nextval('serial');
```

```
nextval
-----
      114
```

Utilizar esta seqüência no comando INSERT:<sup>1</sup>

```
INSERT INTO distribuidores VALUES (nextval('serial'), 'nada');
```

Atualizar o valor da seqüência após executar o comando COPY FROM:

```
BEGIN;
COPY distribuidores FROM 'arquivo_de_entrada';
SELECT setval('serial', max(id)) FROM distribuidores;
END;
```

## Compatibilidade

O comando CREATE SEQUENCE está especificado no SQL:2003. O PostgreSQL está em conformidade com o padrão, com as seguintes exceções:

- A expressão do padrão AS <data type> não é suportada.
- A obtenção do próximo valor é feita usando a função nextval() em vez da expressão do padrão NEXT VALUE FOR.

## Notas

1. Oracle — No Oracle 9i nextval e currval são pseudocolunas e não funções, portanto é usado no comando INSERT, por exemplo, INSERT INTO orders VALUES (orders\_seq.nextval, ... e INSERT INTO order\_items VALUES (orders\_seq.currval, ... Oracle9i SQL Reference - Release 2 (9.2) - Part Number A96540-02 ([http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/sql\\_elements6a.htm](http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/sql_elements6a.htm)) (N. do T.)

# CREATE TABLE

## Nome

CREATE TABLE — cria uma tabela

## Sinopse

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE nome_da_tabela (
    { nome_da_coluna tipo_de_dado [ DEFAULT expressão_padrão ] [ restrição_de_coluna [ ... ] ]
    | restrição_de_tabela
    | LIKE tabela_ancestral [ { INCLUDING | EXCLUDING } DEFAULTS ] } [, ... ]
)
[ INHERITS ( tabela_ancestral [, ... ] ) ]
[ WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE espaço_de_tabelas ]
```

onde *restrição\_de\_coluna* é:

```
[ CONSTRAINT nome_da_restrição ]
{ NOT NULL |
  NULL |
  UNIQUE [ USING INDEX TABLESPACE espaço_de_tabelas ] |
  PRIMARY KEY [ USING INDEX TABLESPACE espaço_de_tabelas ] |
  CHECK (expressão) |
  REFERENCES tabela_referenciada [ ( coluna_referenciada ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE ação ] [ ON UPDATE ação ] }
  [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

e *restrição\_de\_tabela* é:

```
[ CONSTRAINT nome_da_restrição ]
{ UNIQUE ( nome_da_coluna [, ... ] ) [ USING INDEX TABLESPACE espaço_de_tabelas ] |
  PRIMARY KEY ( nome_da_coluna [, ... ] ) [ USING INDEX TABLESPACE espaço_de_tabelas ] |
  CHECK ( expressão ) |
  FOREIGN KEY ( nome_da_coluna [, ... ] )
    REFERENCES tabela_referenciada [ ( coluna_referenciada [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE ação ] [ ON UPDATE ação ] }
  [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

## Descrição

O comando `CREATE TABLE` cria uma tabela, inicialmente vazia, no banco de dados corrente. O usuário que executa o comando se torna o dono da tabela.<sup>1</sup>

Se o nome do esquema for fornecido (por exemplo, `CREATE TABLE meu_esquema.minha_tabela ...`) então a tabela é criada no esquema especificado, senão é criada no esquema corrente. As tabelas temporárias são criadas em um esquema especial e, portanto, o nome do esquema não pode ser especificado ao se criar tabelas temporárias. O nome da tabela deve ser distinto do nome de qualquer outra tabela, sequência, índice ou visão no mesmo esquema.

O comando `CREATE TABLE` também cria, automaticamente, o tipo de dado que representa o tipo composto correspondendo a uma linha da tabela. Portanto, as tabelas não podem ter o mesmo nome de um tipo de dado existente no mesmo esquema.

As cláusulas de restrição opcionais especificam as restrições (ou testes) que as linhas novas ou modificadas devem satisfazer para a operação de inserção ou de modificação ser aceita. Uma restrição é um objeto SQL que ajuda a definir o conjunto de valores válidos para a tabela de várias maneiras.

Existem duas formas para definir restrições: restrições de tabela e restrições de coluna. A restrição de coluna é definida como parte da definição da coluna. A definição da restrição de tabela não é limitada a uma única coluna, podendo incluir mais de uma coluna. Toda restrição de coluna também pode ser escrita como restrição de tabela; a restrição de coluna é somente uma notação conveniente para ser usada quando a restrição afeta apenas uma coluna.<sup>2</sup>

## Parâmetros

TEMPORARY ou TEMP

Se for especificado, a tabela é criada como sendo temporária. As tabelas temporárias são automaticamente removidas no final da sessão ou, opcionalmente, no fim da transação corrente (consulte `ON COMMIT` abaixo). As tabelas permanentes existentes não são visíveis na sessão corrente enquanto existirem tabelas temporárias com o mesmo nome, a não ser que sejam referenciadas por um nome qualificado pelo esquema.<sup>3 4</sup> Todo índice criado em tabela temporária também é temporário.

Opcionalmente, `GLOBAL` ou `LOCAL` podem ser escritos antes de `TEMPORARY` e de `TEMP`. Isto não faz qualquer diferença no PostgreSQL, mas consulte *Compatibilidade*.

*nome\_da\_tabela*

O nome (opcionalmente qualificado pelo esquema) da tabela a ser criada.

*nome\_da\_coluna*

O nome da coluna a ser criada na nova tabela.

*tipo\_de\_dado*

O tipo de dado da coluna. Pode incluir especificadores de matriz (`array`). Para obter informações adicionais sobre os tipos de dado suportados pelo PostgreSQL consulte o Capítulo 8.

DEFAULT *expressão\_padrão*

A cláusula `DEFAULT` atribui um valor de dado padrão para a coluna em cuja definição está presente. O valor pode ser qualquer expressão sem variável (*variable-free*) (subconsultas e referências cruzadas para outras colunas da mesma tabela não são permitidas). O tipo de dado da expressão padrão deve corresponder ao tipo de dado da coluna.

A expressão padrão é utilizada em todas as operações de inserção que não especificam valor para a coluna. Se não houver valor padrão para a coluna, então o valor padrão é o valor nulo.

INHERITS ( *tabela\_ancestral* [, ... ] )

A cláusula opcional `INHERITS` (herda) especifica uma lista de tabelas das quais a nova tabela herda, automaticamente, todas as colunas.

O uso de `INHERITS` cria um relacionamento persistente entre a nova tabela descendente e suas tabelas ancestrais. As modificações de esquema nas tabelas ancestrais normalmente se propagam para as tabelas descendentes também e, por padrão, os dados das tabelas descendentes são incluídos na varredura das tabelas ancestrais.

A cláusula opcional `INHERITS` (herda) especifica uma lista de tabelas das quais a nova tabela herda, automaticamente, todas as colunas. Se o mesmo nome de coluna existir em mais de uma tabela ancestral um erro é relatado, a menos que os tipo de dado das colunas seja o mesmo em todas as tabelas ancestrais. Não havendo conflito, então as colunas duplicadas são unidas para formar uma única coluna da nova tabela. Se a lista de nomes de colunas da nova tabela contiver um nome de coluna que também é herdado, da mesma forma o tipo de dado deve ser o mesmo das colunas herdadas, e a definição das colunas será unida em uma única coluna. Entretanto, declarações de colunas novas e herdadas com o mesmo nome não precisam especificar restrições idênticas: todas as restrições fornecidas em qualquer uma das declarações são unidas, sendo todas aplicadas à nova tabela. Se a nova tabela especificar, explicitamente, um valor padrão para a coluna, este valor padrão substitui o valor padrão das declarações herdadas. Não sendo assim, toda tabela ancestral que especificar um valor padrão para a coluna deve especificar o mesmo valor, ou um erro será relatado.

LIKE *tabela\_ancestral* [ { INCLUDING | EXCLUDING } DEFAULTS ]

A cláusula `LIKE` especifica a tabela da qual a nova tabela copia, automaticamente, todos os nomes de coluna, seus tipos de dado, e restrições de não-nulo.

Ao contrário do `INHERITS`, a nova tabela e a tabela original ficam completamente separadas após o término da criação. Mudanças na tabela original não são aplicadas à nova tabela, e não é possível incluir dados da nova tabela na varredura da tabela original.

As expressões de valor padrão, existentes nas definições das colunas, somente são copiadas quando se especifica `INCLUDING DEFAULTS`. O comportamento padrão é excluir as expressões de valor padrão, fazendo com que todas as colunas da nova tabela tenham como valor padrão o valor nulo.

`WITH OIDS`

`WITHOUT OIDS`

Esta cláusula opcional especifica se as linhas da nova tabela devem possuir OIDs (identificadores de objeto) atribuídos. Se não for especificado nem `WITH OIDS` nem `WITHOUT OIDS`, o valor padrão dependerá do parâmetro de configuração `default_with_oids`; se a nova tabela herdar de alguma tabela que possua OIDs, então `WITH OIDS` é forçado mesmo que o comando contenha `WITHOUT OIDS`.

Se `WITHOUT OIDS` for especificado ou estiver implícito, a nova tabela não armazena OIDs e nenhum OID será atribuído para uma linha inserida na mesma. Normalmente isto é considerado vantajoso, uma vez que reduz o consumo de OIDs e, portanto, adia o reinício do contador de 32-bits do OID. Quando o contador reinicia, não se pode mais assumir que os OIDs sejam únicos, o que os torna muito menos úteis. Além disso, a exclusão dos OIDs da tabela reduz o espaço requerido para armazenar a tabela no disco em 4 bytes por linha (na maioria das máquinas), melhorando um pouco o desempenho.

Para remover os OIDs da tabela após esta ter sido criada deve ser utilizado o comando *ALTER TABLE*.

`CONSTRAINT nome_da_restrição`

Um nome opcional para a restrição de coluna ou de tabela. Se não for especificado, o nome será gerado pelo sistema.

`NOT NULL`

A coluna não pode conter valores nulos.

`NULL`

A coluna pode conter valores nulos. Este é o padrão.

Esta cláusula só está disponível para manter a compatibilidade com bancos de dados SQL fora do padrão. Sua utilização nos novos aplicativos é desencorajada.

`UNIQUE (restrição de coluna)`

`UNIQUE ( nome_da_coluna [ , ... ] ) (restrição de tabela)`

A restrição `UNIQUE` especifica que um grupo de uma ou mais colunas da tabela pode conter apenas valores únicos. O comportamento da restrição de unicidade de tabela é o mesmo da restrição de unicidade de coluna, mas com a capacidade adicional de envolver várias colunas.

Para a finalidade de restrição de unicidade, valores nulos não são considerados iguais.

Cada restrição de unicidade de tabela deve especificar um conjunto de colunas diferente do conjunto de colunas especificado por qualquer outra restrição de unicidade e, também, da chave primária definida para a tabela (Senão, seria apenas a mesma restrição declarada duas vezes).

`PRIMARY KEY (restrição de coluna)`

`PRIMARY KEY ( nome_da_coluna [ , ... ] ) (restrição de tabela)`

A restrição de chave primária especifica que a coluna, ou colunas, da tabela podem conter apenas valores únicos (não duplicados) e não nulos. Tecnicamente a chave primária (`PRIMARY KEY`) é simplesmente uma combinação de unicidade (`UNIQUE`) com não nulo (`NOT NULL`), mas identificar um conjunto de colunas como chave primária também fornece metadados sobre o projeto do esquema, porque chaves primárias têm como consequência permitir que outras tabelas possam depender deste conjunto de colunas como identificador único para linhas.

Somente pode ser especificada uma chave primária para cada tabela, seja como restrição de coluna ou como restrição de tabela.

A restrição de chave primária deve especificar um conjunto de colunas diferente de outro conjunto de colunas especificado por uma restrição de unicidade definido para a mesma tabela.

CHECK (*expressão*)

A cláusula CHECK especifica uma expressão, que produz um resultado booleano, que as linhas novas ou atualizadas devem satisfazer para a operação de inserção ou de atualização ser bem-sucedida. Expressões avaliadas como TRUE ou UNKNOWN são bem-sucedidas. Se alguma linha de uma operação de inserção ou de atualização produzir um resultado FALSE uma exceção de erro é lançada e a inserção ou atualização não altera o banco de dados. Uma restrição de verificação especificada como uma restrição de coluna deve fazer referência somente ao valor desta coluna, enquanto uma expressão que aparece como uma restrição de tabela pode fazer referência a várias colunas.

Atualmente as expressões CHECK não podem conter subconsultas, nem fazer referência a variáveis que não sejam colunas da linha corrente.

REFERENCES *tabela\_referenciada* [ ( *coluna\_referenciada* ) ] [ MATCH *tipo\_de\_correspondência* ] [ ON DELETE *ação* ] [ ON UPDATE *ação* ] (restrição de coluna)

FOREIGN KEY ( *coluna* [, ... ] ) REFERENCES *tabela\_referenciada* [ ( *coluna\_referenciada* [, ... ] ) ] [ MATCH *tipo\_de\_correspondência* ] [ ON DELETE *ação* ] [ ON UPDATE *ação* ] (restrição de tabela)

Estas cláusulas especificam uma restrição de chave estrangeira, que requer que um grupo de uma ou mais colunas da nova tabela deva conter apenas valores correspondendo a valores nas colunas referenciadas de alguma linha da tabela referenciada. Se a *coluna\_referenciada* for omitida, a chave primária da *tabela\_referenciada* será utilizada. As colunas referenciadas devem ser colunas de uma restrição de unicidade ou de chave primária na tabela referenciada.

Os valores inseridos nas colunas que fazem referência são comparados com os valores das colunas referenciadas da tabela referenciada utilizando o tipo de comparação especificado. Existem três tipos de comparação: MATCH FULL, MATCH PARTIAL e MATCH SIMPLE, que também é o padrão. MATCH FULL não permite uma coluna de uma chave estrangeira com várias colunas ser nula, a menos que todas as colunas da chave estrangeira sejam nulas. MATCH SIMPLE permite que algumas colunas da chave estrangeira sejam nulas, enquanto outras colunas da chave estrangeira não são nulas. MATCH PARTIAL ainda não está implementado.

Além disso, quando os dados das colunas referenciadas são modificados, certas ações são realizadas nos dados das colunas desta tabela. A cláusula ON DELETE especifica a ação a ser realizada quando uma linha referenciada da tabela referenciada é excluída. Da mesma forma, a cláusula ON UPDATE especifica a ação a ser realizada quando uma coluna referenciada da tabela referenciada é atualizada para um novo valor. Se a linha for atualizada, mas a coluna referenciada não mudar de valor, nenhuma ação é executada. As ações referenciais fora NO ACTION não podem ser postergadas, mesmo que a restrição seja declarada como postergável (deferrable). São possíveis as seguintes ações para cada cláusula:

## NO ACTION

Produz um erro indicando que a exclusão ou a atualização cria uma violação da restrição de chave estrangeira. Se a restrição for postergada, este erro será produzido em tempo de verificação de restrição se ainda houver alguma linha fazendo referência. Esta é a ação padrão.

## RESTRICT

Produz um erro indicando que a exclusão ou a atualização cria uma violação da restrição de chave estrangeira. É o mesmo que NO ACTION, exceto que a verificação não é postergável.

## CASCADE

Exclui qualquer linha que faça referência à linha excluída, ou atualiza o valor da coluna que faz referência para o novo valor da coluna referenciada, respectivamente.

## SET NULL

Atribui o valor nulo às colunas que fazem referência.

## SET DEFAULT

Atribui o valor padrão às colunas que fazem referência.

Se as colunas referenciadas forem modificadas com frequência, pode ser útil adicionar um índice à coluna da chave estrangeira para que as ações referenciais associadas à coluna da chave estrangeira possam ser realizadas com mais eficiência.

DEFERRABLE

NOT DEFERRABLE

Estas cláusulas controlam se a restrição pode ser postergada. Uma restrição que não pode ser postergada é verificada imediatamente após cada comando. A verificação das restrições postergáveis pode ser adiada para o final da transação (usando o comando *SET CONSTRAINTS*). *NOT DEFERRABLE* é o padrão. Atualmente somente as restrições de chave estrangeira aceitam esta cláusula. Todos os outros tipos de restrição não são postergáveis.

INITIALLY IMMEDIATE

INITIALLY DEFERRED

Se uma restrição for postergável, esta cláusula especifica o instante padrão para verificar a restrição. Se a restrição for *INITIALLY IMMEDIATE*, então será verificada após cada instrução. Este é o padrão. Se a restrição for *INITIALLY DEFERRED*, então será verificada apenas no final da transação. O instante de verificação da restrição pode ser alterado pelo comando *SET CONSTRAINTS*.

ON COMMIT

O comportamento das tabelas temporárias ao término do bloco de transação pode ser controlado utilizando *ON COMMIT*. As três opções são:

PRESERVE ROWS

Nenhuma ação especial é realizada ao término da transação. Este é o comportamento padrão.

DELETE ROWS

Todas as linhas da tabela temporária são excluídas ao término de cada bloco de transação. Essencialmente, um *TRUNCATE* é feito após cada efetivação.

DROP

A tabela temporária é removida ao término do bloco de transação corrente.

*TABLESPACE espaço\_de\_tabelas*

O *espaço\_de\_tabelas* é o nome do espaço de tabelas onde a nova tabela será criada. Se não for especificado será utilizado o *default\_tablespace*, ou o espaço de tabelas padrão do banco de dados se *default\_tablespace* for uma cadeia de caracteres vazia.

*USING INDEX TABLESPACE espaço\_de\_tabelas*

Esta cláusula permite selecionar o espaço de tabelas onde o índice associado à restrição *UNIQUE* ou *PRIMARY KEY* será criado. Se não for especificado será utilizado o *default\_tablespace*, ou o espaço de tabelas padrão do banco de dados se *default\_tablespace* for uma cadeia de caracteres vazia.

## Observações

Não se recomenda usar *OIDs* nos novos aplicativos; é preferível utilizar *SERIAL*, ou outro gerador de seqüências, como chave primária da tabela, sempre que for possível. Entretanto, se o aplicativo já faz uso de *OIDs* para identificar linhas específicas da tabela, é recomendado criar uma restrição de unicidade para a coluna *oid* da tabela, para garantir que os *OIDs* na tabela realmente identificam unicamente uma linha, mesmo após o contador reiniciar. Evite supor que os *OIDs* são únicos entre tabelas; se for necessário um identificador único para todo o banco de dados, use a combinação de *tableoid* (*OID* de tabela) com o *OID* de linha para esta finalidade.

**Dica:** O uso de *WITHOUT OIDS* não é recomendado para tabelas sem chave primária, porque sem um *OID* e sem uma chave de dados única fica difícil identificar uma linha específica.

O PostgreSQL cria, automaticamente, um índice para cada restrição de unicidade e de chave primária para impor a unicidade. Portanto, não é necessário criar explicitamente um índice para as colunas da chave primária (Consulte o comando *CREATE INDEX* para obter mais informações).

As restrições de unicidade e de chave primária não são herdadas na implementação corrente, tornando o comportamento da combinação de herança com restrição de unicidade um tanto disfuncional.<sup>5</sup>

Uma tabela não pode ter mais de 1600 colunas (Na prática o limite efetivo é menor, por causa da restrição do comprimento das tuplas).

## Exemplos

Criar a tabela `filmes` e a tabela `distribuidores`:

```
CREATE TABLE filmes (
    cod_filme    char(5) CONSTRAINT pk_filmes PRIMARY KEY,
    titulo       varchar(40) NOT NULL,
    did          integer NOT NULL,
    data_prod    date,
    tipo         varchar(10),
    duracao      interval hour to minute
);

CREATE TABLE distribuidores (
    did          integer PRIMARY KEY DEFAULT nextval('serial'),
    nome         varchar(40) NOT NULL CHECK (nome <> '')
);
```

Criar uma tabela com uma matriz de 2 dimensões:

```
CREATE TABLE matriz2d_int (
    matriz       int[][]
);
```

Definir uma restrição de unicidade para a tabela `filmes`. Restrições de unicidade usando a sintaxe de restrição de tabela podem ser definidas envolvendo uma ou mais colunas da tabela.

```
CREATE TABLE filmes (
    cod_filme    char(5),
    titulo       varchar(40),
    did          integer,
    data_prod    date,
    tipo         varchar(10),
    duracao      interval hour to minute,
    CONSTRAINT   unq_data_prod UNIQUE(data_prod)
);
```

Definir uma restrição de verificação, usando a sintaxe de restrição de coluna:

```
CREATE TABLE distribuidores (
    did          integer CHECK (did > 100),
    nome         varchar(40)
);
```

Definir uma restrição de verificação, usando a sintaxe de restrição de tabela:

```
CREATE TABLE distribuidores (
    did          integer,
    name         varchar(40)
    CONSTRAINT   chk_dist CHECK (did > 100 AND nome <> '')
);
```

Definir uma restrição de chave primária, usando a sintaxe de restrição de tabela, para a tabela `filmes`. As restrições de chave primária com sintaxe de restrição de tabela podem ser definidas usando uma ou mais colunas da tabela.



```
CREATE TABLE filmes (
    cod_filme    char(5),
    titulo       varchar(40),
    did          integer,
    data_prod    date,
    tipo         varchar(10),
    duracao      interval hour to minute,
    CONSTRAINT pk_filmes PRIMARY KEY(cod_filme,titulo)
);
```

Definir a restrição de chave primária para a tabela distribuidores. Os dois exemplos abaixo são equivalentes, o primeiro utiliza a sintaxe de restrição de tabela, e o segundo utiliza a sintaxe de restrição de coluna.

```
CREATE TABLE distribuidores (
    did          integer,
    name         varchar(40),
    PRIMARY KEY(did)
);
```

```
CREATE TABLE distribuidores (
    did          integer PRIMARY KEY,
    name         varchar(40)
);
```

O comando abaixo atribui uma constante literal como valor padrão para a coluna nome, faz o valor padrão da coluna did ser gerado pela seleção do próximo valor de um objeto de sequência, e faz o valor padrão da coluna data\_mod ser o momento em que a linha foi inserida.

```
CREATE TABLE distribuidores (
    nome         varchar(40) DEFAULT 'Luso Films',
    did          integer DEFAULT nextval('seq_distribuidores'),
    data_mod     timestamp DEFAULT current_timestamp
);
```

Definir duas restrições de coluna NOT NULL na tabela distribuidores, sendo que uma delas recebe um nome fornecido:

```
CREATE TABLE distribuidores (
    did          integer CONSTRAINT nao_nulo NOT NULL,
    nome         varchar(40) NOT NULL
);
```

Definir uma restrição de unicidade para a coluna nome:

```
CREATE TABLE distribuidores (
    did          integer,
    nome         varchar(40) UNIQUE
);
```

O comando acima é equivalente ao mostrado abaixo, especificado como uma restrição de tabela:

```
CREATE TABLE distribuidores (
    did          integer,
    nome         varchar(40),
    UNIQUE(nome)
);
```

Criar a tabela cinemas no espaço de tabelas diskvoll1:

```
CREATE TABLE cinemas (
    id serial,
    name text,
    location text
) TABLESPACE diskvoll1;
```

## Compatibilidade

O comando `CREATE TABLE` está em conformidade com o SQL-92 e com um subconjunto do SQL:1999, com as exceções listadas abaixo.

### Tabelas temporárias

Embora a sintaxe de `CREATE TEMPORARY TABLE` se pareça com a do padrão SQL, o efeito não é o mesmo. No padrão as tabelas temporárias são definidas apenas uma vez, passando a existir automaticamente (começando com um conteúdo vazio) para todas as sessões que necessitarem destas. Em vez disso, o PostgreSQL requer que cada sessão execute seu próprio comando `CREATE TEMPORARY TABLE` para cada tabela temporária a ser utilizada, permitindo que sessões diferentes usem o mesmo nome de tabela temporária para finalidades diferentes, enquanto que a abordagem do padrão restringe todas as instâncias de um determinado nome de tabela temporária terem a mesma estrutura de tabela.

A definição do padrão para o comportamento de tabelas temporárias é amplamente ignorado. O comportamento do PostgreSQL neste ponto é semelhante ao de vários outros bancos de dado SQL.

A distinção feita pelo padrão entre tabelas temporárias globais e locais não está presente no PostgreSQL, uma vez que esta distinção depende do conceito de módulos, que o PostgreSQL não possui. Por motivo de compatibilidade, o PostgreSQL aceita as palavras chave `GLOBAL` e `LOCAL` na declaração da tabela temporária, mas estas não produzem efeito.

A cláusula `ON COMMIT` para as tabelas temporárias também lembra o padrão SQL, mas possui algumas diferenças. Se a cláusula `ON COMMIT` for omitida, o padrão SQL especifica que o comportamento padrão deve ser `ON COMMIT DELETE ROWS`. Entretanto, o comportamento padrão no PostgreSQL é `ON COMMIT PRESERVE ROWS`. A opção `ON COMMIT DROP` não existe no padrão SQL.

### Restrições de verificação de coluna

O padrão SQL estabelece que as restrições de coluna `CHECK` só podem fazer referência à coluna onde estão aplicadas; somente a restrição `CHECK` de tabela pode fazer referência a várias colunas. O PostgreSQL não impõe esta restrição; as restrições de coluna e de tabela são tratadas da mesma maneira.

### “Restrição” `NULL`

A “restrição” `NULL` (na verdade uma não restrição) é uma extensão do PostgreSQL ao padrão SQL incluída para manter a compatibilidade com alguns outros sistemas de banco de dados (e por simetria com a restrição `NOT NULL`). Uma vez que este é o padrão para qualquer coluna, sua presença é desnecessária.

### Herança

Heranças múltiplas por meio da cláusula `INHERITS` é uma extensão do PostgreSQL à linguagem. O SQL:1999 (mas não o SQL-92) define herança única utilizando uma sintaxe diferente e semânticas diferentes. O estilo de herança do SQL:1999 ainda não é suportado pelo PostgreSQL.

### Identificadores de Objeto (Object IDs)

O conceito de OIDs (identificadores de objeto) do PostgreSQL não é padrão.

### Tabelas sem coluna

O PostgreSQL permite a criação de tabelas sem colunas (por exemplo, `CREATE TABLE foo();`). Isto é uma extensão ao padrão SQL, que não permite tabelas com zero coluna. As tabelas sem coluna não são muito úteis, mas se não forem permitidas criam um caso especial para o comando `ALTER TABLE DROP COLUMN` e, por isso, parece mais simples ignorar esta restrição contida na especificação.

### Espaços de tabelas

O conceito de espaços de tabelas do PostgreSQL não faz parte do padrão. Portanto, as cláusulas `TABLESPACE` e `USING INDEX TABLESPACE` são extensões.

## Consulte também

ALTER TABLE, DROP TABLE, CREATE TABLESPACE

## Notas

1. A *tabela* é uma coleção ordenada de uma ou mais colunas e uma coleção não ordenada de zero ou mais linhas. Cada linha possui, para cada coluna, exatamente um valor do tipo de dado desta coluna. (ISO-ANSI Working Draft) Framework (SQL/Framework), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)

2. A *restrição de tabela* é uma restrição de integridade associada a uma única tabela base.

A restrição de tabela é uma entre *restrição de unicidade*, *restrição de chave primária*, *restrição referencial* ou *restrição de verificação*.

A restrição de unicidade especifica uma ou mais colunas da tabela como *colunas únicas*. A restrição de unicidade é satisfeita se, e somente se, não houverem duas linhas da tabela com os mesmos valores não-nulos nas colunas únicas.

A restrição de chave primária é a restrição de unicidade que especifica PRIMARY KEY. A restrição de chave primária é satisfeita se, e somente se, não houverem duas linhas da tabela com os mesmos valores não-nulos nas colunas únicas, e nenhum dos valores da coluna ou colunas especificadas for o valor nulo.

A restrição referencial especifica uma ou mais colunas como *colunas que fazem referência*, e as *colunas referenciadas* correspondentes em alguma (não necessariamente distinta) tabela base, referida como *tabela referenciada*. Estas colunas referenciadas são colunas únicas de alguma restrição de unicidade da tabela referenciada. A restrição referencial está sempre satisfeita se, para toda linha da tabela que faz referência, os valores das colunas que fazem referência são iguais àqueles das colunas referenciadas correspondentes de alguma linha da tabela referenciada. Entretanto, se estiverem presentes valores nulos a satisfação da integridade referencial depende do tratamento especificado para os nulos (conhecido como o *tipo de correspondência*).

Podem ser especificadas *ações referenciais* para determinar que alterações devem ser feitas na tabela que faz referência se, de outra forma, uma alteração na tabela referenciada causasse a violação da restrição referencial.

Uma restrição de verificação de tabela especifica uma *condição de procura*. A restrição é violada se o resultado da condição de procura for falso para qualquer linha da tabela (mas não se for desconhecido).

(ISO-ANSI Working Draft) Framework (SQL/Framework), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)

3. DB2 — Quando um programa ou usuário declara uma tabela temporária, a única maneira de referenciá-la no SQL é através da utilização do qualificador SESSION. Se o qualificador SESSION não for utilizado, o DB2 tenta encontrar a tabela utilizando o esquema corrente. DB2® Universal Database V8 for Linux, UNIX, and Windows Database Administration Certification Guide, 5th Edition (<http://www.phptr.com/title/0130463612>), George Baklarz e Bill Wong, Series IBM Press, Prentice Hall Professional Technical Reference, 2003, pág. 193. (N. do T.)
4. SQL Server 2000 — As tabelas temporárias são semelhantes às tabelas permanentes, exceto que as tabelas temporárias são armazenadas em tempdb, e removidas automaticamente quando não estão mais em uso. Os dois tipos de tabela temporária, local e global, diferem um do outro em nome, visibilidade e disponibilidade. A tabela temporária local possui um caractere jogo-da-velha (#) como o primeiro caractere de seu nome; é enxergada apenas pela conexão corrente com o usuário; e é removida quando o usuário se desconecta da instância do SQL Server 2000. A tabela global temporária possui dois caracteres jogo-da-velha (##) como os primeiros caracteres de seu nome; é enxergada por qualquer usuário após ser criada; e é removida quando todos os usuários fazendo referência à tabela se desconectam da instância do SQL Server. SQL Server Books Online (N. do T.)
5. disfunção — dificuldade ou problema de funcionamento. PRIBERAM - Língua Portuguesa On-Line (<http://www.priberam.pt/dlpo/dlpo.aspx>) (N. do T.)

# CREATE TABLE AS

## Nome

CREATE TABLE AS — cria uma tabela a partir dos resultados de uma consulta

## Sinopse

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE nome_da_tabela [ (nome_da_coluna
[, ...] ) ] [ [ WITH | WITHOUT ] OIDS ]
    AS comando
```

## Descrição

O comando CREATE TABLE AS cria uma tabela e a carrega com dados computados pelo comando SELECT, ou por um comando EXECUTE que executa um comando SELECT preparado. As colunas da tabela possuem os nomes e tipos de dado associados às colunas da saída do comando SELECT (exceto que é possível mudar os nomes das colunas fornecendo uma lista explícita de novos nomes de colunas).

O comando CREATE TABLE AS possui alguma semelhança com a criação de uma visão, mas na realidade é bastante diferente: este comando cria a nova tabela e executa a consulta apenas uma vez para fazer a carga inicial da nova tabela. A nova tabela não acompanha as mudanças posteriores ocorridas nas tabelas de origem da consulta. Contrastando com este comportamento, a visão executa novamente o comando SELECT que a define sempre que uma consulta é realizada.

## Parâmetros

GLOBAL ou LOCAL

Ignorado por compatibilidade. Consulte CREATE TABLE para obter detalhes.

TEMPORARY ou TEMP

Se for especificado, a tabela é criada como uma tabela temporária. Consulte o comando CREATE TABLE para obter mais detalhes.

*nome\_da\_tabela*

O nome (opcionalmente qualificado pelo esquema) da tabela a ser criada.

*nome\_da\_coluna*

O nome da coluna na nova tabela. Se os nomes das colunas não forem fornecidos, são obtidos a partir dos nomes das colunas produzidas pela consulta. Se a tabela for criada a partir de um comando EXECUTE, a lista de nomes de colunas não pode ser especificada.

WITH OIDS

WITHOUT OIDS

Esta cláusula opcional especifica se a tabela criada pelo comando CREATE TABLE AS deve incluir os OIDs. Se não for especificada nenhuma das formas desta cláusula, é utilizado o valor do parâmetro de configuração default\_with\_oids.

*comando*

Um comando de consulta (ou seja, um comando SELECT ou um comando EXECUTE que executa um comando SELECT preparado). Consulte SELECT ou EXECUTE, respectivamente, para obter a descrição da sintaxe permitida.

## Observações

Este comando é funcionalmente equivalente ao SELECT INTO mas é preferível, porque é menos propenso a ser confundido com outros usos da sintaxe do comando SELECT ... INTO. Além disso, o comando CREATE TABLE AS oferece um superconjunto das funcionalidades oferecidas pelo SELECT INTO.

Antes do PostgreSQL 8.0, o comando `CREATE TABLE AS` sempre incluía os OIDs nas tabelas produzidas. A partir do PostgreSQL 8.0, o comando `CREATE TABLE AS` permite ao usuário especificar explicitamente se os OIDs devem ser incluídos. Se a presença dos OIDs não for especificada explicitamente, é usada a variável de configuração `default_with_oids`. Embora o valor padrão corrente seja `TRUE`, este valor padrão pode ser mudado no futuro. Portanto, os aplicativos que requerem que os OIDs sejam criados na tabela pelo comando `CREATE TABLE AS` devem especificar `WITH OIDS` para garantir a compatibilidade com as versões futuras do PostgreSQL.

## Exemplos

Criar a tabela `filmes_recentes` consistindo apenas das entradas recentes da tabela `filmes`:

```
CREATE TABLE filmes_recentes AS
  SELECT * FROM filmes WHERE data_prod >= '2002-01-01';
```

## Compatibilidade

O comando `CREATE TABLE AS` está especificado no padrão SQL:2003. Existem algumas pequenas diferenças entre a definição do comando no SQL:2003 e sua implementação no PostgreSQL:

- O padrão requer parênteses em torno da cláusula de subconsulta; no PostgreSQL estes parênteses são opcionais.
- O padrão define a cláusula `ON COMMIT`, que não está atualmente implementada pelo PostgreSQL.
- O padrão define a cláusula `WITH DATA`, que não está atualmente implementada pelo PostgreSQL.

## Consulte também

*CREATE TABLE, EXECUTE, SELECT, SELECT INTO*

# CREATE TABLESPACE

## Nome

CREATE TABLESPACE — cria um espaço de tabelas

## Sinopse

```
CREATE TABLESPACE nome_do_espaco_de_tabelas [ OWNER nome_do_usuario ] LOCATION 'diretorio'
```

## Descrição

O comando CREATE TABLESPACE registra um novo espaço de tabelas para todo o agrupamento. O nome do espaço de tabelas deve ser distinto do nome de qualquer outro espaço de tabelas existente no agrupamento de bancos de dados.

O espaço de tabelas permite aos superusuários definirem um local alternativo no sistema de arquivos onde os arquivos de dados contendo os objetos do banco de dados (tais como tabelas e índices) podem residir.

Um usuário com os privilégios apropriados pode passar o *nome\_do\_espaco\_de\_tabelas* para os comandos CREATE DATABASE, CREATE TABLE, CREATE INDEX e ADD CONSTRAINT para que os arquivos de dados destes objetos sejam armazenados no espaço de tabelas especificado.

## Parâmetros

*nome\_do\_espaco\_de\_tabelas*

O nome do espaço de tabelas a ser criado. O nome não pode começar por pg\_, uma vez que estes nomes são reservados para espaços de tabelas do sistema.

*nome\_do\_usuario*

O nome do usuário que será o dono do espaço de tabelas. Se for omitido, o padrão é o usuário que está executando o comando. Somente os superusuários podem criar espaços de tabelas, mas podem atribuir a propriedade dos espaços de tabelas a usuários comuns.

*diretorio*

O diretório a ser usado pelo espaço de tabelas. O diretório deve estar vazio e deve pertencer ao usuário de sistema do PostgreSQL. O diretório deve ser especificado por um nome de caminho absoluto.

## Observações

Os espaços de tabelas são suportados apenas nos sistemas que suportam vínculos simbólicos.

## Exemplos

Criar o espaço de tabelas dbspace em /data/dbs:

```
CREATE TABLESPACE dbspace LOCATION '/data/dbs';
```

Criar o espaço de tabelas indexspace em /data/indexes pertencendo ao usuário genevieve:

```
CREATE TABLESPACE indexspace OWNER genevieve LOCATION '/data/indexes';
```

## Compatibilidade

O comando CREATE TABLESPACE é uma extensão do PostgreSQL.

## Consulte também

CREATE DATABASE, CREATE TABLE, CREATE INDEX, DROP TABLESPACE, ALTER TABLESPACE

# CREATE TRIGGER

## Nome

CREATE TRIGGER — cria um gatilho

## Sinopse

```
CREATE TRIGGER nome { BEFORE | AFTER } { evento [ OR ... ] }  
ON tabela [ FOR [ EACH ] { ROW | STATEMENT } ]  
EXECUTE PROCEDURE nome_da_função ( argumentos )
```

## Descrição

O comando `CREATE TRIGGER` cria um gatilho. O gatilho fica associado à tabela especificada e executa a função especificada *nome\_da\_função* quando determinados eventos ocorrem.<sup>1 2 3</sup>

O gatilho pode ser especificado para disparar antes de tentar realizar a operação na linha (antes das restrições serem verificadas e o comando `INSERT`, `UPDATE` ou `DELETE` ser tentado), ou após a operação estar completa (após as restrições serem verificadas e o `INSERT`, `UPDATE` ou `DELETE` ter completado). Se o gatilho for disparado antes do evento, o gatilho pode evitar a operação para a linha corrente, ou modificar a linha sendo inserida (para as operações de `INSERT` e `UPDATE` somente). Se o gatilho for disparado após o evento, todas as mudanças, incluindo a última inserção, atualização ou exclusão, são “visíveis” para o gatilho.

Se o gatilho estiver marcado como `FOR EACH ROW` então é chamado uma vez para cada linha modificada pela operação. Por exemplo, um comando `DELETE` afetando 10 linhas faz com que todos os gatilhos `ON DELETE` da relação de destino sejam chamados 10 vezes, uma vez para cada linha excluída. Por outro lado, um gatilho marcado como `FOR EACH STATEMENT` somente executa uma vez para uma determinada operação, a despeito de quantas linhas sejam modificadas; em particular, uma operação que não modifica nenhuma linha ainda assim resulta na execução de todos os gatilhos `FOR EACH STATEMENT` aplicáveis.

Se vários gatilhos do mesmo tipo estão definidos para o mesmo evento, estes são disparados na ordem alfabética de seus nomes<sup>4</sup>.

O `SELECT` não modifica nenhuma linha e, portanto, não é possível criar gatilhos para `SELECT`. Regras e visões são mais apropriadas neste caso.

Consulte o Capítulo 32 para obter informações adicionais sobre gatilhos.

## Parâmetros

*nome*

O nome a ser dado ao novo gatilho, devendo ser distinto do nome de qualquer outro gatilho para a mesma tabela.

`BEFORE`

`AFTER`

Determina se a função é chamada antes ou depois do evento.

*evento*

Um entre `INSERT`, `UPDATE` ou `DELETE`; especifica o evento que dispara o gatilho. Vários eventos podem ser especificados utilizando `OR`.

*tabela*

O nome (opcionalmente qualificado pelo esquema) da tabela que o gatilho se destina.

FOR EACH ROW  
FOR EACH STATEMENT

Especifica se o procedimento do gatilho deve ser disparado uma vez para cada linha afetada pelo evento do gatilho, ou apenas uma vez para a declaração SQL. Se nenhum dos dois for especificado, `FOR EACH STATEMENT` é usado por padrão.

*nome\_da\_função*

Uma função fornecida pelo usuário, declarada como não recebendo nenhum argumento e retornando o tipo `trigger`, que é executada quando o gatilho dispara.

*argumentos*

Uma lista opcional de argumentos, separados por vírgula, passada para a função quando o gatilho é executado. Os argumentos são constantes cadeia de caracteres literais. Também podem ser escritos nomes simples e constantes numéricas, mas estes são convertidos em cadeias de caracteres. Por favor, verifique como os argumentos do gatilho são acessados dentro da função, na descrição da linguagem de implementação da função de gatilho; pode ser diferente dos argumentos das função normais.

## Observações

Para poder criar um gatilho em uma tabela, o usuário deve possuir o privilégio `TRIGGER` na tabela.

Nas versões do PostgreSQL anteriores a 7.3, era necessário declarar as funções dos gatilhos como retornando o tipo `opaque` em vez de `trigger`. Para permitir a carga das cópias de segurança antigas, o comando `CREATE TRIGGER` aceita funções declaradas como retornando `opaque`, mas mostra uma mensagem e muda para `trigger` o tipo retornado declarado pela função.

Deve ser utilizado o comando `DROP TRIGGER` para remover um gatilho.

## Exemplos

A Seção 32.4 contém um exemplo completo.

## Compatibilidade

A declaração `CREATE TRIGGER` do PostgreSQL implementa um subconjunto do padrão SQL:1999 (O padrão SQL-92 não trata de gatilhos). As seguintes funcionalidades estão faltando:

- O padrão SQL:1999 permite que os gatilhos sejam disparados pela atualização de colunas específicas (por exemplo, `AFTER UPDATE OF col1, col2`).
- O padrão SQL:1999 permite definir outros nomes (*aliases*) para as linhas e tabelas “velhas” e “novas” a serem utilizados na definição da ação do gatilho (por exemplo, `CREATE TRIGGER ... ON nome_da_tabela REFERENCING OLD ROW AS algum_nome NEW ROW AS outro_nome ...`). Uma vez que o PostgreSQL permite que os procedimentos dos gatilhos sejam escritos em qualquer linguagem definida pelo usuário, o acesso aos dados é tratado na forma específica da linguagem.
- O PostgreSQL somente permite a execução de uma função definida pelo usuário para a ação do gatilho. O padrão SQL:1999 permite a execução de vários outros comandos SQL, como o `CREATE TABLE`, para a ação do gatilho. Esta limitação é fácil de ser superada criando uma função definida pelo usuário para executar os comandos desejados.

O SQL:1999 especifica que os vários gatilhos devem ser disparados na ordem da data de criação. O PostgreSQL usa a ordem dos nomes, que foi considerada mais conveniente para se trabalhar.

A capacidade de especificar várias ações para um único gatilho utilizando `OR` é uma extensão do PostgreSQL ao padrão SQL.

## Consulte também

`CREATE FUNCTION`, `ALTER TRIGGER`, `DROP TRIGGER`



## Notas

1. O *gatilho*, embora não seja definido como um componente da tabela base, é um objeto associado a uma única tabela base. O gatilho especifica o *evento de gatilho*, o *momento de ação do gatilho*, e uma ou mais *ações engatilhadas*.  
 O evento de gatilho especifica que ação na tabela base deverá causar as ações engatilhadas. O evento de gatilho é um entre INSERT, DELETE e UPDATE.  
 O momento de ação do gatilho especifica se a ação do gatilho será efetuada BEFORE (antes) ou AFTER (após) o evento do gatilho.  
 A ação engatilhada é um procedimento SQL ou BEGIN ATOMIC, seguido por uma ou mais <declaração de procedimento SQL>, seguido por END.  
 (ISO-ANSI Working Draft) Framework (SQL/Framework), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
2. Oracle — Os gatilhos de banco de dados permitem definir e obrigar regras de integridade, mas os gatilhos de banco de dados não são a mesma coisa que restrições de integridade. Entre outras coisas, o gatilho de banco de dados não verifica os dados já carregados na tabela. Portanto, é altamente recomendado que o uso de gatilhos de banco de dados seja feito somente quanto as regras de integridade referencial não puderem ser obrigadas através de restrições de integridade. Introduction to the Oracle Server  
 ([http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96524/c01\\_02intro.htm#45920](http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96524/c01_02intro.htm#45920)) (N. do T.)
3. IBM — No Centro de Controle do DB2 8.1 *trigger* é traduzido como *disparador*. (N. do T.)
4. Oracle — Embora os gatilhos de tipos diferentes sejam disparados em uma ordem específica, não há garantia que os gatilhos do mesmo tipo para o mesmo comando sejam disparados em uma ordem específica. Por exemplo, pode ser que os gatilhos BEFORE row para o mesmo comando UPDATE não sejam disparados sempre na mesma ordem. Os aplicativos devem ser projetados de forma que não dependam da ordem de disparo de vários gatilhos do mesmo tipo. Triggers (<http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96524/c18trigs.htm#3585>) (N. do T.)

# CREATE TYPE

## Nome

CREATE TYPE — cria um tipo de dado

## Sinopse

```
CREATE TYPE nome AS
    ( nome_do_atributo tipo_de_dado [, ... ] )

CREATE TYPE nome (
    INPUT = função_de_entrada,
    OUTPUT = função_de_saída
    [ , RECEIVE = função_de_recepção ]
    [ , SEND = função_de_envio ]
    [ , ANALYZE = função_de_análise ]
    [ , INTERNALLENGTH = { comprimento_interno | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alinhamento ]
    [ , STORAGE = armazenamento ]
    [ , DEFAULT = padrão ]
    [ , ELEMENT = elemento ]
    [ , DELIMITER = delimitador ]
)
```

## Descrição

O comando `CREATE TYPE` registra um novo tipo de dado para uso no banco de dados corrente. O usuário que define o tipo se torna o seu dono.

Se o nome do esquema for fornecido, então o tipo será criado no esquema especificado, senão será criado no esquema corrente. O nome do tipo deve ser distinto do nome de qualquer tipo ou domínio existente no mesmo esquema (como as tabelas possuem tipos de dado associados, o nome do tipo também deve ser distinto do nome de qualquer tabela existente no mesmo esquema).

## Tipos Compostos

A primeira forma de `CREATE TYPE` cria um tipo composto. O tipo composto é especificado por uma lista de nomes de atributo e tipos de dado. É essencialmente a mesma coisa que o tipo de uma linha de tabela, mas o uso de `CREATE TYPE` evita a necessidade de criar uma tabela real quando tudo o que se deseja é apenas definir um tipo. Um tipo composto autônomo é útil como argumento ou tipo retornado por uma função.

## Tipos Base

A segunda forma do comando `CREATE TYPE` cria um novo tipo base (tipo escalar). Os parâmetros podem estar em qualquer ordem, e não apenas na ordem mostrada acima, e a maior parte é opcional. Requer o registro de duas ou mais funções (utilizando `CREATE FUNCTION`) antes de definir o tipo. As funções de suporte *função\_de\_entrada* e *função\_de\_saída* são requeridas, enquanto as funções *função\_de\_recepção*, *função\_de\_envio* e *função\_de\_análise* são opcionais. Geralmente estas funções precisam ser codificadas em C ou em outra linguagem de baixo nível.

A *função\_de\_entrada* converte a representação textual externa do tipo na representação interna utilizada pelos operadores e funções definidos para o tipo. A *função\_de\_saída* realiza a transformação inversa. A função de entrada pode ser declarada como recebendo um argumento do tipo `cstring`, ou recebendo três argumentos dos tipos `cstring`, `oid` e `integer`. O primeiro argumento é o texto de entrada como uma cadeia de caracteres C, o segundo argumento é o OID do tipo do elemento no caso de ser um tipo matriz (ou o próprio OID do tipo para tipos compostos), e o terceiro é o `typmod` da coluna de destino, se for conhecido (será passado -1 se não for conhecido). A função de entrada deve retornar um valor do próprio tipo de dado. A função de saída pode ser declarada como recebendo um argumento do novo tipo de

dado, ou como recebendo dois argumentos dos quais o segundo é do tipo `oid`. O segundo argumento é novamente o OID do tipo do elemento da matriz para tipos matriz, ou o OID do tipo para tipos compostos. A função de saída deve retornar o tipo `cstring`.

A função *função\_de\_recepção* opcional converte a representação binária externa do tipo para a representação interna. Se esta função não for fornecida, o tipo não pode participar de entrada binária. A representação binária deve ser escolhida para ser de baixo custo converter para a forma interna, e ao mesmo tempo razoavelmente portátil (Por exemplo, os tipos de dado inteiro padrão utilizam a ordem de byte de rede como sendo a representação binária externa, enquanto a representação interna está na ordem de byte nativa da máquina). A função de recepção deve realizar uma verificação adequada para garantir que o valor seja válido. A função de recepção pode ser declarada como recebendo um argumento do tipo `internal`, ou dois argumentos dos tipos `internal` e `oid`. Deve retornar um valor do próprio tipo de dado (O primeiro argumento é um ponteiro para o `buffer StringInfo`, que armazena a cadeia de bytes recebida; o segundo argumento opcional é o OID do tipo do elemento no caso de ser um tipo matriz, ou o próprio OID do tipo para um tipo composto. Da mesma maneira, a função *função\_de\_envio* opcional converte da representação interna para a representação binária externa. Se esta função não for fornecida, o tipo não pode participar de saída binária. A função de envio pode ser declarada como recebendo um argumento do novo tipo de dado, ou como recebendo dois argumentos dos quais o segundo é do tipo `oid`. O segundo argumento é novamente o OID do tipo do elemento da matriz para tipos matriz, ou o OID do tipo para tipos compostos. A função de envio deve retornar o tipo `bytea`.

Neste ponto podemos estar querendo saber como as funções de entrada e de saída podem ser declaradas possuindo resultados ou argumentos do novo tipo, se devem ser criadas antes que o novo tipo possa ser criado. A resposta é que a função de entrada deve ser criada primeiro, depois a função de saída (e as funções binárias de entrada e de saída, se for desejado) e, finalmente, o tipo de dado. O PostgreSQL vê primeiro o nome do novo tipo de dado como tipo retornado pela função de entrada, e cria um tipo de “abrigo”, que é simplesmente uma entrada no catálogo do sistema, e vincula a definição da função de entrada ao tipo de abrigo. Da mesma maneira, as outras funções são vinculadas ao (agora já existente) tipo de abrigo. Por último, o comando `CREATE TYPE` substitui a entrada de abrigo com a definição completa do tipo, e o novo tipo poderá ser usado.

A função *função\_de\_análise* opcional realiza a coleta de estatísticas específicas do tipo, para as colunas com este tipo de dado. Por padrão, o comando `ANALYZE` tenta obter as estatísticas usando os operadores “igual” e “menor-que” do tipo, se houver uma classe de operadores `b-tree` padrão para o tipo. Para os tipos não escalares, este comportamento provavelmente não será adequado e, portanto, pode ser substituído especificando-se uma função de análise personalizada. Esta função deve ser declarada como recebendo um único argumento do tipo `internal`, e retornando um resultado do tipo `boolean`. A API detalhada para a função de análise está descrita em `src/include/commands/vacuum.h`.

Enquanto os detalhes da representação interna do novo tipo são conhecidos somente pelas funções de entrada e de saída e por outras funções criadas pelo usuário para trabalhar com o tipo, existem várias propriedades da representação interna que devem ser declaradas para o PostgreSQL. A mais importante destas é o *comprimento\_interno*. Os tipos de dado base podem ser de comprimento fixo e, neste caso, o *comprimento\_interno* é um inteiro positivo, ou de comprimento variável, indicado pela definição do *comprimento\_interno* como `VARIABLE` (Internamente isto é representado definindo `typlen` como `-1`). A representação interna de todos os tipos de comprimento variável devem começar por um inteiro de 4 bytes fornecendo o comprimento total deste valor do tipo.

O sinalizador opcional `PASSEDBYVALUE` indica que os valores deste tipo de dado são passados por valor, e não por referência. Não é possível passar por valor tipos cuja representação interna seja maior do que o tamanho do tipo `Datum` (4 bytes na maioria das máquinas, e 8 bytes em poucas).

O parâmetro *alinhamento* especifica o alinhamento de armazenamento requerido por este tipo de dado. Os valores permitidos igualam-se ao alinhamento nas fronteiras de 1, 2, 4 ou 8 bytes. Deve ser observado que os tipos de comprimento variável devem ter um alinhamento de pelo menos 4, porque contêm, necessariamente, um `int4` como seu primeiro componente.

O parâmetro *armazenamento* permite selecionar estratégias de armazenamento para tipos de dado de comprimento variável (somente é permitido `plain` para os tipos de comprimento fixo). `plain` especifica que os dados deste tipo são sempre armazenado em-linha e não comprimidos. `extended` especifica que primeiro o sistema tentará comprimir um valor de dado longo e, depois, movê-lo para fora da linha da tabela principal se ainda permanecerem muito longo. `external` permite mover os valores para fora da tabela principal, mas o sistema não tentará comprimi-los. `main` permite a compressão, mas desencoraja mover o valor para fora da tabela principal (os itens de dado com esta estratégia de armazenamento só são movidos para fora da tabela principal se não houver outra maneira de fazer a linha caber, mas têm mais preferência para serem mantidos na tabela principal do que os itens `extended` e `external`).

Pode ser especificado um valor padrão, quando se deseja que as colunas com este tipo de dado tenham como padrão algo diferente do valor nulo. O valor padrão é especificado pela palavra chave `DEFAULT`; este padrão pode ser substituído por uma cláusula `DEFAULT` explícita anexada a uma determinada coluna.

Para indicar que o tipo é uma matriz, deve ser especificado o tipo dos elementos da matriz utilizando a palavra chave `ELEMENT`. Por exemplo, para definir uma matriz de inteiros de 4 bytes (`int4`), deve ser especificado `ELEMENT = int4`. Mais detalhes sobre tipos matriz são mostrados abaixo.

Para indicar o delimitador a ser usado entre os valores na representação externa das matrizes deste tipo, pode-se definir *delimitador* como um caractere específico. O delimitador padrão é a vírgula (,). Deve ser observado que o delimitador está associado com o tipo do elemento da matriz, e não com a própria matriz.

## Tipos matriz

Sempre que um tipo de dado base definido pelo usuário é criado, o PostgreSQL cria automaticamente um tipo matriz (array) associado, cujo nome consiste no nome do tipo base prefixado pelo caractere sublinhado. O analisador compreende esta convenção de nome, e traduz as solicitações para colunas do tipo `foo[]` em solicitações para o tipo `_foo`. O tipo matriz criado implicitamente é de comprimento variável e usa as funções de entrada e saída nativas `array_in` e `array_out`.

Pode ser perguntado, com razão, por que existe a opção `ELEMENT` se o sistema produz o tipo matriz correto automaticamente. O único caso onde é útil utilizar `ELEMENT` é quando se constrói um tipo de comprimento fixo que é internamente uma matriz de componentes idênticos, e deseja-se que estes componentes sejam acessados diretamente por índices, além de qualquer outra operação que se planeje fornecer para este tipo como um todo. Por exemplo, o tipo `name` permite que os elementos `char` que o constituem sejam acessados desta forma. Um tipo `point` 2-D pode permitir que os dois números que o compõe sejam acessados como `point[0]` e `point[1]`. Deve ser observado que esta praticidade somente funciona em tipos de comprimento fixo cuja forma interna seja exatamente uma seqüência de campos idênticos de comprimento fixo. Para permitir o uso de índices, um tipo de comprimento variável deve possuir a representação interna generalizada usada por `array_in` e `array_out`. Por razões históricas (ou seja, claramente errado mas muito tarde para mudar) os índices dos tipos matriz de comprimento fixo começam por zero, em vez de começar por um como no caso matrizes de comprimento variável.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) do tipo a ser criado.

*nome\_do\_atributo*

O nome de um atributo (coluna) para o tipo composto.

*tipo\_de\_dado*

O nome de um tipo de dado existente que será uma coluna do tipo composto.

*função\_de\_entrada*

O nome da função que converte os dados da forma textual externa para a forma interna.

*função\_de\_saída*

O nome da função que converte os dados da forma interna do tipo para a forma textual externa.

*função\_de\_recepção*

O nome da função que converte os dados da forma binária externa do tipo para a forma interna.

*função\_de\_envio*

O nome da função que converte os dados da forma interna do tipo para a forma binária externa.

*função\_de\_análise*

O nome da função que realiza as análises estatísticas para o tipo de dado.

*comprimento\_interno*

Uma constante numérica que especifica o comprimento em bytes da representação interna do novo tipo. O padrão é assumir como sendo de comprimento variável.

*alinhamento*

O alinhamento de armazenamento requerido por este tipo de dado. Se for especificado, deve ser `char`, `int2`, `int4` ou `double`; o padrão é `int4`.

*armazenamento*

A estratégia de armazenamento para este tipo de dado. Se for especificado, deve ser `plain`, `external`, `extended` ou `main`; o padrão é `plain`.

*padrão*

O valor padrão para este tipo de dado. Se for omitido, o padrão é nulo.

*elemento*

O tipo sendo criado é uma matriz (`array`); especifica o tipo dos elementos da matriz.

*delimitador*

O caractere delimitador a ser usado entre os valores nas matrizes (`arrays`) feitas deste tipo.

## Observações

Os nomes de tipo definidos pelo usuário não podem começar pelo caractere sublinhado (`_`), e só podem ter 62 caracteres de comprimento (ou, de modo geral, `NAMEDATALEN-2`, em vez dos `NAMEDATALEN-1` caracteres permitidos para os outros nomes). Os nomes de tipo começados por sublinhado são reservados para os nomes de tipo matriz criados internamente.

Nas versões do PostgreSQL anteriores a 7.3, era costume evitar criar um tipo “abrigo” substituindo as referências à frente da função para o nome do tipo usando o pseudotipo `opaque`. Os argumentos e resultados `cstring` também tinham que ser declarados como `opaque` antes da versão 7.3. Para permitir a carga de cópias de segurança antigas, o comando `CREATE TYPE` aceita funções declaradas como retornando `opaque`, mas mostra uma mensagem e muda a declaração da função para usar o tipo correto.

## Exemplos

Este exemplo cria um tipo composto e o utiliza na definição de uma função:

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

Este exemplo cria o tipo de dado base `box` e, em seguida, o utiliza na definição da tabela:

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

Se a estrutura interna de `box` fosse uma matriz de quatro elementos `float4`, poderia ter sido usado

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

o que permitiria acessar o valor de um componente de `box` através de índice. Fora isso, o tipo se comporta do mesmo modo que o anterior.

Este exemplo cria um tipo objeto grande e o utiliza na definição de uma tabela:

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

Outros exemplos, incluindo funções de entrada e de saída, podem ser encontrados no Capítulo 31.

## Compatibilidade

Este comando `CREATE TYPE` é uma extensão do PostgreSQL. Existe um comando `CREATE TYPE` no SQL:1999 e posterior que é bastante diferente nos detalhes.

## Consulte também

*CREATE FUNCTION, DROP TYPE, ALTER TYPE*

# CREATE USER

## Nome

CREATE USER — cria uma conta de usuário do banco de dados

## Sinopse

```
CREATE USER nome [ [ WITH ] opção [ ... ] ]
```

onde *opção* pode ser:

```
    SYSID id_do_usuario
| CREATEDB | NOCREATEDB
| CREATEUSER | NOCREATEUSER
| IN GROUP nome_do_grupo [ , ... ]
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'senha'
| VALID UNTIL 'data_e_hora'
```

## Descrição

O comando CREATE USER adiciona um novo usuário ao agrupamento de bancos de dados do PostgreSQL. Consulte o Capítulo 17 e o Capítulo 19 para obter informações adicionais sobre o gerenciamento de usuários e autenticação. Apenas os superusuários do banco de dados podem usar este comando.

## Parâmetros

*nome*

O nome do usuário.

*id\_do\_usuario*

A cláusula SYSID pode ser utilizada para escolher o identificador de usuário do PostgreSQL do novo usuário. Normalmente não é necessário, mas pode ser útil se for necessário recriar o dono de um objeto que ficou órfão.

Se não for especificado, será utilizado por padrão o maior identificador de usuário atribuído acrescido de um (com mínimo de 100).

CREATEDB

NOCREATEDB

Estas cláusulas definem a permissão para o usuário criar banco de dados. Se CREATEDB for especificado, o usuário sendo definido terá permissão para criar seus próprios bancos de dados. Se NOCREATEDB for especificado, nega-se ao usuário a permissão para criar banco de dados. Se esta cláusula for omitida, NOCREATEDB será utilizado por padrão.

CREATEUSER

NOCREATEUSER

Estas cláusulas determinam se o usuário pode ou não criar novos usuários. CREATEUSER também torna o usuário um superusuário, o qual pode substituir todas as restrições de acesso. Se não for especificada, NOCREATEUSER é o padrão.

*nome\_do\_grupo*

O nome de um grupo existente onde o usuário será incluído como um novo membro. Podem ser especificados nomes de vários grupos.

*senha*

Define a senha do usuário. Se não se pretende utilizar autenticação por senha esta opção pode ser omitida, mas o usuário não poderá mais se conectar se for decidido mudar para autenticação por senha. A senha poderá ser definida ou mudada posteriormente através do comando ALTER USER.

ENCRYPTED  
UNENCRYPTED

Estas palavras chave controlam se a senha é armazenada criptografada, ou não, nos catálogos do sistema; se nenhuma das duas for especificada, o comportamento padrão é determinado pelo parâmetro de configuração `password_encryption`. Se a cadeia de caracteres da senha já estiver criptografada no formato MD5, então a cadeia de caracteres é armazenada como está, independentemente de `ENCRYPTED` ou `UNENCRYPTED` ser especificado (porque o sistema não pode descriptografar a cadeia de caracteres criptografada contendo a senha). Esta funcionalidade permite a restauração de senhas criptografadas efetuadas por uma operação de `dump/restore`.

Deve ser observado que clientes antigos podem não possuir suporte para o mecanismo de autenticação MD5, necessário para trabalhar com senhas armazenadas criptografadas.

*data\_e\_hora*

A cláusula `VALID UNTIL` define uma data e hora após a qual a senha do usuário não é mais válida. Se esta cláusula for omitida, a conta será válida para sempre.

## Observações

Deve ser usado o comando `ALTER USER` para mudar os atributos de um usuário, e `DROP USER` para remover um usuário. Deve se usado `ALTER GROUP` para adicionar ou remover usuários de grupos.

O PostgreSQL inclui o programa `createuser` que possui a mesma funcionalidade do `CREATE USER` (na verdade, chama este comando), mas pode ser executado a partir da linha de comando.

A cláusula `VALID UNTIL` define uma data de expiração para a senha apenas, e não para a conta do usuário *per se*. Em particular, a obediência à data de expiração não é imposta ao se conectar utilizando um método de autenticação não baseado em senha.

## Exemplos

Criar um usuário sem senha:

```
CREATE USER jonas;
```

Criar um usuário com senha:

```
CREATE USER manuel WITH PASSWORD 'jw8s0F4';
```

Criar um usuário com uma senha válida até o fim de 2004. Após o primeiro segundo de 2005 a senha não será mais válida.

```
CREATE USER miriam WITH PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

Criar uma conta onde o usuário pode criar bancos de dados:

```
CREATE USER manuel WITH PASSWORD 'jw8s0F4' CREATEDB;
```

## Compatibilidade

O comando `CREATE USER` é uma extensão do PostgreSQL. O padrão SQL deixa a definição de usuários para a implementação.

## Consulte também

`ALTER USER`, `DROP USER`, `createuser`



# CREATE VIEW

## Nome

`CREATE VIEW` — cria uma visão

## Sinopse

```
CREATE [ OR REPLACE ] VIEW nome [ ( nome_da_coluna [, ...] ) ] AS consulta
```

## Descrição

O comando `CREATE VIEW` cria uma visão. A visão não é materializada fisicamente. Em vez disso, a consulta é executada toda vez que a visão é referenciada em uma consulta.<sup>1 2</sup>

O comando `CREATE OR REPLACE VIEW` é semelhante, mas se já existir uma visão com o mesmo nome então esta é substituída. Uma visão somente pode ser substituída por uma nova consulta que produza um conjunto idêntico de colunas (ou seja, colunas com mesmos nomes e tipos de dado).

Se for fornecido o nome do esquema (por exemplo, `CREATE VIEW meu_esquema.minha_visao ...`) então a visão será criada no esquema especificado, senão será criada no esquema corrente. O nome da visão deve ser distinto do nome de qualquer outra visão, tabela, seqüência ou índice no mesmo esquema.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) da visão a ser criada.

*nome\_da\_coluna*

Uma lista opcional de nomes a serem usados para as colunas da visão. Se não for fornecida, os nomes das colunas são determinados a partir da consulta.

*consulta*

Uma consulta (ou seja, um comando `SELECT`) que gera as colunas e linhas da visão.

Consulte o comando `SELECT` para obter informações adicionais sobre as consultas válidas.

## Observações

Atualmente, as visões são somente para leitura: o sistema não permite inserção, atualização ou exclusão em uma visão. É possível obter o efeito de uma visão atualizável criando regras que reescrevem as inserções, etc. na visão como ações apropriadas em outras tabelas. Para obter informações adicionais consulte o comando `CREATE RULE`.

Deve ser utilizado o comando `DROP VIEW` para remover uma visão.

Tome cuidado para que os nomes e os tipos das colunas da visão sejam atribuídos da maneira desejada. Por exemplo,

```
CREATE VIEW vista AS SELECT 'Hello World';
```

é ruim por dois motivos: o nome padrão da coluna é `?column?`, e o tipo de dado padrão da coluna é `unknown`. Se for desejado um literal cadeia de caracteres no resultado da visão deve ser utilizado algo como

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

O acesso às tabelas referenciadas pela visão é determinado pelas permissões do dono da visão. Entretanto, as funções chamadas pela visão são tratadas da mesma maneira como se tivessem sido chamadas diretamente pela consulta que utiliza a visão. Portanto, o usuário da visão deve possuir permissão para chamar todas as funções utilizadas pela visão.

## Exemplos

Criar uma visão mostrando todos os filmes de comédia:

```
CREATE VIEW comedias AS
  SELECT *
  FROM filmes
  WHERE tipo = 'Comédia';
```

## Compatibilidade

O padrão SQL especifica algumas funcionalidades adicionais para o comando CREATE VIEW:

```
CREATE VIEW nome [ ( coluna [, ...] ) ]
  AS consulta
  [ WITH [ CASCADE | LOCAL ] CHECK OPTION ]
```

As cláusulas opcionais para o comando SQL completo são:

CHECK OPTION

Esta opção está associada às visões atualizáveis. Todos os comandos INSERT e UPDATE na visão são verificados para garantir que os dados satisfazem às condições que definem a visão (ou seja, os novos dados devem ser visíveis através da visão). Se as condições não forem satisfeitas, a atualização será rejeitada.

LOCAL

Verifica a integridade nesta visão.

CASCADE

Verifica a integridade nesta visão e em todas as visões dependentes. CASCADE é assumido se nem CASCADE nem LOCAL forem especificados.

O comando CREATE OR REPLACE VIEW é uma extensão do PostgreSQL à linguagem.

## Consulte também

*DROP VIEW*

## Notas

1. A *visão* (estritamente, a *definição da visão*) é uma consulta nomeada, que pode para muitas finalidades ser utilizada da mesma maneira que uma tabela base. Seu valor é o resultado da avaliação da consulta. (ISO-ANSI Working Draft) Framework (SQL/Framework), August 2003, ISO/IEC JTC 1/SC 32, 25-jul-2003, ISO/IEC 9075-2:2003 (E) (N. do T.)
2. IBM — No Centro de Controle do DB2 8.1 *view* é traduzido como *visualização*. (N. do T.)

# DEALLOCATE

## Nome

DEALLOCATE — remove um comando preparado

## Sinopse

```
DEALLOCATE [ PREPARE ] nome
```

## Descrição

O comando `DEALLOCATE` é utilizado para remover um comando SQL previamente preparado. Se o comando preparado não for removido explicitamente, então será removido quando a sessão terminar.

Para obter informações adicionais sobre comandos preparados consulte o comando *PREPARE*.

## Parâmetros

`PREPARE`

Esta palavra chave é ignorada.

*nome*

O nome do comando preparado a ser removido.

## Compatibilidade

O padrão SQL inclui o comando `DEALLOCATE`, mas apenas para uso na linguagem SQL incorporada (embedded).

## Consulte também

*EXECUTE*, *PREPARE*

# DECLARE

## Nome

DECLARE — define um cursor

## Sinopse

```
DECLARE nome [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]  
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR consulta  
    [ FOR { READ ONLY | UPDATE [ OF coluna [, ...] ] } ]
```

## Descrição

O comando `DECLARE` permite o usuário criar cursores, que podem ser utilizados para trazer, de cada vez, um pequeno número de linhas de uma consulta grande. Os cursores podem retornar dados tanto no formato texto quanto binário usando o comando `FETCH`.

Cursores normais retornam dados no formato texto, o mesmo produzido pelo comando `SELECT`. Como os dados são armazenados nativamente no formato binário, o sistema necessita realizar uma conversão para gerar o formato texto. Como a informação chega no formato texto, o aplicativo cliente pode precisar convertê-la para o formato binário para manipulá-la. Além disso, dados no formato texto geralmente possuem um tamanho maior que no formato binário. Os cursores binários retornam os dados na representação binária, que pode ser manipulada mais facilmente. Entretanto, se o objetivo é exibir os dados na forma de texto, trazê-los na forma de texto reduz um pouco o esforço no lado cliente.

Como exemplo, se uma consulta retornar o valor “um” de uma coluna com tipo de dado inteiro, será recebida a cadeia de caracteres 1 com o cursor padrão, enquanto que com um cursor binário será retornado um campo de 4 bytes contendo a representação interna do valor (na ordem de byte “big-endian”<sup>1</sup>).

Os cursores binários devem ser usados com cuidado. Muitos aplicativos, incluindo o `psql`, não estão preparados para tratar cursores binários e esperam que os dados cheguem no formato texto.

**Nota:** Quando o aplicativo cliente utiliza o protocolo “consulta estendida” (`extended query`) para executar o comando `FETCH`, a mensagem `Bind` do protocolo especifica se os dados devem retornar no formato texto ou binário. Esta escolha substitui a forma como o cursor foi definido. Por isso o conceito de cursor binário fica obsoleto ao se utilizar o protocolo “consulta estendida” — todo cursor pode ser tratado tanto como texto ou binário.

## Parâmetros

*nome*

O nome do cursor a ser criado.

`BINARY`

Faz o cursor retornar os dados no formato binário em vez do formato texto.

`INSENSITIVE`

Indica que os dados trazidos pelo cursor não devem ser afetados pelas atualizações feitas nas tabelas subjacentes ao cursor, enquanto o cursor existir. No PostgreSQL, todos os cursores são insensíveis; atualmente esta palavra chave não produz efeito, estando presente por motivo de compatibilidade com o padrão SQL.

`SCROLL`

`NO SCROLL`

`SCROLL` (rolar) especifica que o cursor pode ser utilizado para trazer linhas de uma maneira não seqüencial (por exemplo, para trás). Dependendo da complexidade do plano de execução da consulta, especificar `SCROLL` pode impor uma penalidade de desempenho no tempo de execução da consulta. `NO SCROLL` especifica que o cursor não pode ser utilizado para trazer linhas de uma maneira não seqüencial.

WITH HOLD  
WITHOUT HOLD

WITH HOLD especifica que o cursor pode continuar sendo utilizado após a transação que o criou ter sido efetivada com sucesso. WITHOUT HOLD especifica que o cursor não pode ser utilizado fora da transação que o criou. Se nem WITHOUT HOLD nem WITH HOLD for especificado, WITHOUT HOLD é o padrão.

#### consulta

O comando SELECT que produz as linhas a serem retornadas pelo cursor. Consulte o comando *SELECT* para obter informações adicionais sobre consultas válidas.

FOR READ ONLY  
FOR UPDATE

FOR READ ONLY indica que o cursor será utilizado no modo somente-leitura. FOR UPDATE indica que o cursor será utilizado para atualizar tabelas. Uma vez que atualizações por cursor não são suportadas pelo PostgreSQL no momento, especificar FOR UPDATE causa uma mensagem de erro, e especificar FOR READ ONLY não produz efeito.

#### coluna

Colunas a serem atualizadas pelo cursor. Uma vez que atualizações por cursor não são suportadas pelo PostgreSQL no momento, a cláusula FOR UPDATE provoca uma mensagem de erro.

As palavras chave BINARY, INSENSITIVE e SCROLL podem estar em qualquer ordem.

## Observações

A menos que WITH HOLD seja especificado, o cursor criado por este comando poderá ser utilizado apenas dentro da transação corrente. Portanto, DECLARE sem WITH HOLD não possui utilidade fora do bloco de transação: o cursor existe apenas até o término da instrução. Por esse motivo, o PostgreSQL relata um erro se este comando for utilizado fora de um bloco de transação. Devem ser utilizados os comandos *BEGIN*, *COMMIT* e *ROLLBACK* para definir um bloco de transação.

Se WITH HOLD for especificado, e a transação que criou o cursor for efetivada com sucesso, o cursor pode continuar sendo acessado pelas transações seguintes na mesma sessão (Mas se a transação que o criou for interrompida, o cursor é removido). O cursor criado com WITH HOLD é fechado quando um comando CLOSE explícito é executado para o cursor, ou quando a sessão termina. Na implementação atual, as linhas representadas por um cursor mantido são copiadas para um arquivo temporário, ou para uma área de memória, para permanecerem disponíveis para as transações seguintes.

A opção SCROLL deve ser especificada ao se definir um cursor utilizado para trazer para trás. Isto é requerido pelo padrão SQL. Entretanto, para manter a compatibilidade com as versões anteriores, o PostgreSQL permite trazer para trás sem a opção SCROLL, se o plano da consulta do cursor for simples o suficiente para que nenhum trabalho extra seja necessário para isto. Entretanto, aconselha-se aos desenvolvedores de aplicativos a não confiar na utilização de trazer para trás a partir de um cursor que não tenha sido criado com a opção de SCROLL. Se NO SCROLL for especificado, então trazer para trás não é permitido em qualquer caso.

O padrão SQL somente trata de cursores na linguagem SQL incorporada. O servidor PostgreSQL não implementa o comando OPEN para cursores; o cursor é considerado aberto ao ser declarado. Entretanto o ECPG, o pré-processador do PostgreSQL para a linguagem SQL incorporada, suporta as convenções de cursor do padrão SQL, incluindo as que envolvem as instruções DECLARE e OPEN.

## Exemplos

Para declarar um cursor:

```
DECLARE liahona CURSOR FOR SELECT * FROM filmes;
```

Veja no comando *FETCH* mais exemplos de utilização de cursor.

## Compatibilidade

O padrão SQL permite cursores somente na linguagem SQL incorporada e nos módulos. O PostgreSQL permite que o cursor seja utilizado interativamente.

O padrão SQL permite que os cursores atualizem os dados das tabelas. Todos os cursores do PostgreSQL são somente para leitura.

Os cursores binários são uma extensão do PostgreSQL.

## Consulte também

*CLOSE, FETCH, MOVE*

## Notas

1. *big-endian* — uma arquitetura de computadores na qual, em uma representação numérica com vários bytes, o byte mais significativo possui o menor endereço (a palavra é armazenada com o “maior-fim-primeiro”). FOLDOC - Free On-Line Dictionary of Computing (<http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=big-endian>) (N. do T.)

# DELETE

## Nome

DELETE — exclui linhas de uma tabela

## Sinopse

```
DELETE FROM [ ONLY ] tabela [ WHERE condição ]
```

## Descrição

O comando DELETE exclui da tabela especificada as linhas que satisfazem a cláusula WHERE. Se a cláusula WHERE estiver ausente, o efeito produzido é a exclusão de todas as linhas da tabela. O resultado é uma tabela válida, porém vazia.

**Dica:** O comando *TRUNCATE* é uma extensão do PostgreSQL que fornece um mecanismo mais rápido para excluir todas as linha da tabela.

Por padrão, o comando DELETE exclui linhas da tabela especificada e de todas as suas descendentes. Se for desejado excluir linhas apenas da tabela especificada, deve ser utilizada a cláusula ONLY.

É necessário possuir o privilégio DELETE na tabela para excluir linhas da mesma, assim como o privilégio SELECT para todas as tabelas cujos valores são lidos pela *condição*.

## Parâmetros

*tabela*

O nome (opcionalmente qualificado pelo esquema) de uma tabela existente.

*condição*

Uma expressão de valor, que retorna um valor do tipo boolean, que determina as linhas a serem excluídas.

## Saídas

Ao terminar bem-sucedido, o comando DELETE retorna uma linha de fim de comando na forma

```
DELETE contador
```

O *contador* é o número de linhas excluídas. Se *contador* for igual a 0, então não há linhas correspondendo à *condição* (isto não é considerado um erro).

## Observações

O PostgreSQL permite que se faça referência a colunas de outras tabelas na condição WHERE. Por exemplo, para excluir todos os filmes produzidos por um determinado produtor pode ser utilizado:

```
DELETE FROM filmes
WHERE filmes.id_produto = produtores.id_produto AND produtores.nome = 'foo';
```

Essencialmente o que acontece neste comando é uma junção entre as tabelas filmes e produtores, com todas as linhas de filmes juntadas com sucesso sendo marcadas para exclusão. Esta sintaxe não é padrão. Uma forma de se escrever mais de acordo com o padrão seria:

```
DELETE FROM filmes
WHERE id_produto IN (SELECT id_produto FROM produtores WHERE nome = 'foo');
```

Em alguns casos o estilo junção é mais fácil de ser escrito ou mais rápido de executar do que o estilo subseleção. Uma objeção ao estilo junção é que não existe uma lista explícita de quais tabelas estão sendo utilizadas, o que torna este estilo propenso a erros; também não pode tratar autojunções.

## Exemplos

Excluir todos os filmes, exceto os musicais:

```
DELETE FROM filmes WHERE tipo <> 'Musical';
```

Esvaziar a tabela filmes:

```
DELETE FROM filmes;
```

## Compatibilidade

Este comando está em conformidade com o padrão SQL, exceto que capacidade de fazer referência a outras tabelas na cláusula `WHERE` é uma extensão do PostgreSQL.



# DROP AGGREGATE

## Nome

DROP AGGREGATE — remove uma função de agregação

## Sinopse

```
DROP AGGREGATE nome ( tipo ) [ CASCADE | RESTRICT ]
```

## Descrição

O comando DROP AGGREGATE remove uma função de agregação existente. Para executar este comando o usuário corrente deve ser o dono da função de agregação.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) de uma função de agregação existente.

*tipo*

O tipo de dado do argumento da função de agregação, ou \* se a função aceitar qualquer tipo de dado.

CASCADE

Exclui automaticamente os objetos que dependem da função de agregação.

RESTRICT

Não permite remover a função de agregação caso existam objetos que dependam da mesma. Este é o padrão.

## Exemplos

Para remover a função de agregação `minha_media` para o tipo de dado `integer`:

```
DROP AGGREGATE minha_media(integer);
```

## Compatibilidade

Não existe o comando DROP AGGREGATE no padrão SQL.

## Consulte também

*ALTER AGGREGATE*, *CREATE AGGREGATE*

# DROP CAST

## Nome

DROP CAST — remove uma conversão de tipo de dado

## Sinopse

```
DROP CAST (tipo_de_origem AS tipo_de_destino) [ CASCADE | RESTRICT ]
```

## Descrição

O comando DROP CAST remove uma conversão de tipo de dado previamente definida.

Para poder remover uma conversão de tipo de dado é necessário ser o dono do tipo de dado de origem ou de destino. São os mesmos privilégios necessários para criar uma conversão de tipo de dado.

## Parâmetros

*tipo\_de\_origem*

O nome do tipo de dado de origem da conversão.

*tipo\_de\_destino*

O nome do tipo de dado de destino da conversão.

CASCADE

RESTRICT

Estas palavras chave não produzem nenhum efeito, porque não existem dependências para conversões de tipo de dado.

## Exemplos

Para remover a conversão do tipo text para o tipo int:

```
DROP CAST (text AS int);
```

## Compatibilidade

O comando DROP CAST está em conformidade com o padrão SQL.

## Consulte também

*CREATE CAST*

# DROP CONVERSION

## Nome

DROP CONVERSION — remove uma conversão de codificação

## Sinopse

```
DROP CONVERSION nome [ CASCADE | RESTRICT ]
```

## Descrição

O comando DROP CONVERSION remove uma conversão de codificação definida anteriormente. É necessário ser o dono da conversão para poder removê-la.

## Parâmetros

*nome*

O nome da conversão de codificação. O nome da conversão pode ser qualificado pelo esquema.

CASCADE

RESTRICT

Estas palavras chave não produzem nenhum efeito, porque não há dependências em conversões de codificação.

## Exemplos

Para remover a conversão de codificação chamada `minha_conversao`:

```
DROP CONVERSION minha_conversao;
```

## Compatibilidade

Não existe o comando DROP CONVERSION no padrão SQL.

## Consulte também

*ALTER CONVERSION, CREATE CONVERSION*

# DROP DATABASE

## Nome

`DROP DATABASE` — remove um banco de dados

## Sinopse

`DROP DATABASE nome`

## Descrição

O comando `DROP DATABASE` remove um banco de dados. Remove as entradas no catálogo para o banco de dados e remove o diretório contendo os dados. Pode ser executado apenas pelo dono do banco de dados. Também, não pode ser executado se você ou alguma outra pessoa estiver conectado ao banco de dados a ser removido; se conecte ao `template1` ou a qualquer outro banco de dados para executar este comando.

O comando `DROP DATABASE` não pode ser desfeito. Utilize com cuidado!

## Parâmetros

*nome*

O nome do banco de dados a ser removido.

## Observações

O comando `DROP DATABASE` não pode ser executado dentro de um bloco de transação.

Este comando não pode ser executado enquanto conectado ao banco de dados de destino. Portanto, é mais conveniente utilizar o aplicativo *dropdb*, que é um script englobando este comando.

## Compatibilidade

Não existe o comando `DROP DATABASE` no padrão SQL.

## Consulte também

`CREATE DATABASE`

# DROP DOMAIN

## Nome

DROP DOMAIN — remove um domínio

## Sinopse

```
DROP DOMAIN nome [, ...] [ CASCADE | RESTRICT ]
```

## Descrição

O comando DROP DOMAIN remove um domínio. Somente o dono do domínio pode removê-lo.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) de um domínio existente.

CASCADE

Remove automaticamente os objetos que dependem do domínio (como colunas de tabelas).

RESTRICT

Não permite remover o domínio caso existam objetos que dependam do mesmo. Este é o padrão.

## Exemplos

Para remover o domínio box:

```
DROP DOMAIN box;
```

## Compatibilidade

Este comando está em conformidade com o padrão SQL.

## Consulte também

*CREATE DOMAIN*

# DROP FUNCTION

## Nome

DROP FUNCTION — remove uma função

## Sinopse

```
DROP FUNCTION nome ( [ tipo [, ...] ] ) [ CASCADE | RESTRICT ]
```

## Descrição

O comando DROP FUNCTION remove a definição de uma função existente. Para executar este comando o usuário deve ser o dono da função. Os tipos de dado dos argumentos da função devem ser especificados, porque várias funções diferentes podem existir com o mesmo nome e listas de argumentos diferentes.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) de uma função existente.

*tipo*

O tipo de dado de um argumento da função.

CASCADE

Remove automaticamente os objetos que dependem da função (como operadores e gatilhos).

RESTRICT

Não permite remover a função caso existam objetos que dependam da mesma. Este é o padrão.

## Exemplos

Este comando remove a função que calcula a raiz quadrada:

```
DROP FUNCTION sqrt(integer);
```

## Compatibilidade

Existe um comando DROP FUNCTION definido no padrão SQL, mas não é compatível com este comando.

## Consulte também

CREATE FUNCTION, ALTER FUNCTION

# DROP GROUP

## Nome

`DROP GROUP` — remove um grupo de usuários

## Sinopse

`DROP GROUP nome`

## Descrição

O comando `DROP GROUP` remove o grupo especificado. Os usuários no grupo não são removidos.

## Parâmetros

*nome*

O nome de um grupo existente.

## Observações

Não se aconselha remover um grupo que tenha permissões concedidas para objetos. Atualmente isto não é exigido, mas é provável que as versões futuras do PostgreSQL verifiquem este erro.

## Exemplos

Para remover um grupo:

```
DROP GROUP engenharia;
```

## Compatibilidade

Não existe o comando `DROP GROUP` no padrão SQL.

## Consulte também

`ALTER GROUP`, `CREATE GROUP`

# DROP INDEX

## Nome

DROP INDEX — remove um índice

## Sinopse

```
DROP INDEX nome [ , ... ] [ CASCADE | RESTRICT ]
```

## Descrição

O comando `DROP INDEX` remove do sistema de banco de dados um índice existente. Para executar este comando é necessário ser o dono do índice.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) do índice a ser removido.

CASCADE

Remove automaticamente os objetos que dependem do índice.

RESTRICT

Não permite remover o índice caso existam objetos que dependam do mesmo. Este é o padrão.

## Exemplos

O comando a seguir remove o índice `idx_titulo`:

```
DROP INDEX idx_titulo;
```

## Compatibilidade

O comando `DROP INDEX` é uma extensão do PostgreSQL à linguagem. O padrão SQL não trata de índices.

## Consulte também

`CREATE INDEX`



# DROP LANGUAGE

## Nome

DROP LANGUAGE — remove uma linguagem procedural

## Sinopse

```
DROP [ PROCEDURAL ] LANGUAGE nome [ CASCADE | RESTRICT ]
```

## Descrição

O comando DROP LANGUAGE remove a definição da linguagem procedural previamente registrada chamada *nome*.

## Parâmetros

*nome*

O nome de uma linguagem procedural existente. Para manter a compatibilidade com as versões anteriores, o nome pode estar entre apóstrofes ( ' ).

CASCADE

Remove automaticamente os objetos que dependem da linguagem (como as funções escritas nesta linguagem).

RESTRICT

Não permite remover a linguagem caso existam objetos que dependam da mesma. Este é o padrão.

## Exemplos

O comando mostrado abaixo remove a linguagem procedural `plsample`:

```
DROP LANGUAGE plsample;
```

## Compatibilidade

Não existe o comando DROP LANGUAGE no padrão SQL.

## Consulte também

*ALTER LANGUAGE*, *CREATE LANGUAGE*, *droplang*

# DROP OPERATOR

## Nome

DROP OPERATOR — remove um operador

## Sinopse

```
DROP OPERATOR nome ( { tipo_à_esquerda | NONE } , { tipo_à_direita | NONE } ) [ CASCADE | RESTRICT ]
```

## Descrição

O comando DROP OPERATOR remove do sistema de banco de dados um operador existente. Para executar este comando é necessário ser o dono do operador.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) de um operador existente.

*tipo\_à\_esquerda*

O tipo de dado do operando à esquerda do operador; deve ser escrito NONE se o operador não possuir operando à esquerda.

*tipo\_à\_direita*

O tipo de dado do operando à direita do operador; deve ser escrito NONE se o operador não possuir operando à direita.

CASCADE

Remove automaticamente os objetos que dependem do operador.

RESTRICT

Não permite remover o operador caso existam objetos que dependam do mesmo. Este é o padrão.

## Exemplos

Remover o operador de potência  $a^b$  para o tipo integer:

```
DROP OPERATOR ^ (integer, integer);
```

Remover o operador de complemento bit-a-bit unário esquerdo  $\sim b$  para o tipo bit:

```
DROP OPERATOR ~ (none, bit);
```

Remover o operador de fatorial unário direito  $x!$  para o tipo bigint:

```
DROP OPERATOR ! (bigint, none);
```

## Compatibilidade

Não existe o comando DROP OPERATOR no padrão SQL.

## Consulte também

CREATE OPERATOR, ALTER OPERATOR

# DROP OPERATOR CLASS

## Nome

DROP OPERATOR CLASS — remove uma classe de operadores

## Sinopse

```
DROP OPERATOR CLASS nome USING método_de_índice [ CASCADE | RESTRICT ]
```

## Descrição

O comando DROP OPERATOR CLASS remove uma classe de operadores existente. É necessário ser o dono da classe de operadores para executar este comando.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) de uma classe de operadores existente.

*método\_de\_índice*

O nome do método de acesso de índice para o qual a classe de operadores se destina.

CASCADE

Remove automaticamente os objetos que dependem da classe de operadores.

RESTRICT

Não permite remover a classe de operadores caso existam objetos que dependam da mesma. Este é o comportamento padrão.

## Exemplos

Remover a classe de operadores B-tree widget\_ops:

```
DROP OPERATOR CLASS widget_ops USING btree;
```

Este comando não é bem-sucedido quando existe algum índice utilizando a classe de operadores. Deve ser especificado CASCADE para remover estes índices junto com a classe de operadores.

## Compatibilidade

Não existe o comando DROP OPERATOR CLASS no padrão SQL.

## Consulte também

ALTER OPERATOR CLASS, CREATE OPERATOR CLASS

# DROP RULE

## Nome

DROP RULE — remove uma regra de reescrita

## Sinopse

```
DROP RULE nome ON relação [ CASCADE | RESTRICT ]
```

## Descrição

O comando DROP RULE remove uma regra de reescrita.

## Parâmetros

*nome*

O nome da regra a ser removida.

*relação*

O nome (opcionalmente qualificado pelo esquema) da tabela ou da visão à qual a regra se aplica.

CASCADE

Remove automaticamente os objetos que dependem da regra.

RESTRICT

Não permite remover a regra caso existam objetos que dependam da mesma. Este é o padrão.

## Exemplos

Para remover a regra de reescrita nova\_regra:

```
DROP RULE nova_regra ON minha_tabela;
```

## Compatibilidade

Não existe o comando DROP RULE no padrão SQL.

## Consulte também

*CREATE RULE*

# DROP SCHEMA

## Nome

DROP SCHEMA — remove um esquema

## Sinopse

```
DROP SCHEMA nome [ , ... ] [ CASCADE | RESTRICT ]
```

## Descrição

O comando DROP SCHEMA remove esquemas do banco de dados.

O esquema somente pode ser removido pelo seu dono ou por um superusuário. Deve ser observado que o dono pode remover o esquema (e, portanto, todos os objetos que este contém), mesmo que não seja o dono de alguns objetos contidos no esquema.

## Parâmetros

*nome*

O nome do esquema.

CASCADE

Remove automaticamente os objetos (tabelas, funções, etc.) contidos no esquema.

RESTRICT

Não permite remover o esquema caso este contenha algum objeto. Este é o padrão.

## Exemplos

Para remover do banco de dados o esquema `meu_esquema` junto com todos os objetos que este contém:

```
DROP SCHEMA meu_esquema CASCADE;
```

## Compatibilidade

O comando DROP SCHEMA está em conformidade total com o padrão SQL, exceto que o padrão permite apenas remover um esquema por comando.

## Consulte também

*ALTER SCHEMA*, *CREATE SCHEMA*

# DROP SEQUENCE

## Nome

`DROP SEQUENCE` — remove uma seqüência

## Sinopse

```
DROP SEQUENCE nome [ , ... ] [ CASCADE | RESTRICT ]
```

## Descrição

O comando `DROP SEQUENCE` remove geradores de números seqüenciais.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) da seqüência.

`CASCADE`

Remove automaticamente os objetos que dependem da seqüência.

`RESTRICT`

Não permite remover a seqüência caso existam objetos que dependam da mesma. Este é o padrão.

## Exemplos

Para remover a seqüência `serial`:

```
DROP SEQUENCE serial;
```

## Compatibilidade

O comando `DROP SEQUENCE` está em conformidade com o padrão SQL:2003, exceto que o padrão permite a remoção de apenas uma seqüência por comando.

## Consulte também

`CREATE SEQUENCE`

# DROP TABLE

## Nome

DROP TABLE — remove uma tabela

## Sinopse

```
DROP TABLE nome [ , ... ] [ CASCADE | RESTRICT ]
```

## Descrição

O comando `DROP TABLE` remove tabelas do banco de dados. Somente o criador pode destruir a tabela. Para deixar uma tabela sem linhas, sem destruí-la, deve ser usado o comando `DELETE`.

O comando `DROP TABLE` sempre remove todos os índices, regras, gatilhos e restrições existentes na tabela de destino. Entretanto, para remover uma tabela referenciada por uma visão ou por uma restrição de chave estrangeira de outra tabela, deve ser especificado `CASCADE` (`CASCADE` remove inteiramente a visão dependente, mas no caso da restrição de chave estrangeira somente a chave estrangeira é removida, e não a outra tabela inteiramente).

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) da tabela a ser removida.

`CASCADE`

Remove automaticamente os objetos que dependem da tabela (como as visões).

`RESTRICT`

Não permite remover a tabela caso existam objetos que dependam da mesma. Este é o padrão.

## Exemplos

Remover duas tabelas, `filmes` e `distribuidores`:

```
DROP TABLE filmes, distribuidores;
```

## Compatibilidade

Este comando está em conformidade com o padrão SQL, exceto que o padrão permite a remoção de apenas uma tabela por comando.

## Consulte também

`ALTER TABLE`, `CREATE TABLE`

# DROP TABLESPACE

## Nome

DROP TABLESPACE — remove um espaço de tabelas

## Sinopse

```
DROP TABLESPACE nome_do_espaco_de_tabelas
```

## Descrição

O comando DROP TABLESPACE remove do sistema um espaço de tabelas.

O espaço de tabelas somente pode ser removido pelo seu dono ou por um superusuário. Devem ser removidos do espaço de tabelas todos os objetos de banco de dados antes que este possa ser removido. É possível que objetos de outros bancos de dados ainda residam no espaço de tabelas, mesmo que nenhum objeto do banco de dados corrente esteja utilizando o espaço de tabelas.

## Parâmetros

*nome\_do\_espaco\_de\_tabelas*

O nome do espaço de tabelas.

## Exemplos

Para remover do sistema o espaço de tabelas minhas\_coisas:

```
DROP TABLESPACE minhas_coisas;
```

## Compatibilidade

O comando DROP TABLESPACE é uma extensão do PostgreSQL.

## Consulte também

*CREATE TABLESPACE, ALTER TABLESPACE*



# DROP TRIGGER

## Nome

DROP TRIGGER — remove um gatilho

## Sinopse

```
DROP TRIGGER nome ON tabela [ CASCADE | RESTRICT ]
```

## Descrição

O comando `DROP TRIGGER` remove uma definição de gatilho existente. Para executar este comando, o usuário corrente deve ser o dono da tabela para a qual o gatilho está definido.

## Parâmetros

*nome*

O nome do gatilho a ser removido.

*tabela*

O nome (opcionalmente qualificado pelo esquema) da tabela para a qual o gatilho está definido.

CASCADE

Remove automaticamente os objetos que dependem do gatilho.

RESTRICT

Não permite remover o gatilho caso existam objetos que dependam do mesmo. Este é o padrão.

## Exemplos

Remover o gatilho `se_dist_existe` da tabela `filmes`:

```
DROP TRIGGER se_dist_existe ON filmes;
```

## Compatibilidade

O comando `DROP TRIGGER` do PostgreSQL é incompatível com o padrão SQL. No padrão SQL os nomes de gatilhos não são locais às tabelas e, portanto, o comando é simplesmente `DROP TRIGGER nome`.

## Consulte também

`CREATE TRIGGER`

# DROP TYPE

## Nome

`DROP TYPE` — remove um tipo de dado

## Sinopse

```
DROP TYPE nome [ , ... ] [ CASCADE | RESTRICT ]
```

## Descrição

O comando `DROP TYPE` remove um tipo de dado definido pelo usuário. Somente o dono do tipo de dado pode removê-lo.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) do tipo de dado a ser removido.

`CASCADE`

Remove automaticamente os objetos que dependem do tipo de dado (como colunas de tabelas, funções, operadores, etc.).

`RESTRICT`

Não permite remover o tipo de dado caso existam objetos que dependam do mesmo. Este é o padrão.

## Exemplos

Para remover o tipo de dado `box`:

```
DROP TYPE box;
```

## Compatibilidade

Este comando é semelhante ao comando correspondente do padrão SQL, mas deve ser observado que o comando `CREATE TYPE` e os mecanismos de extensão de tipo de dado do PostgreSQL são diferentes do padrão SQL.

## Consulte também

*CREATE TYPE*, *ALTER TYPE*

# DROP USER

## Nome

`DROP USER` — remove uma conta de usuário do banco de dados

## Sinopse

```
DROP USER nome
```

## Descrição

O comando `DROP USER` remove o usuário especificado. Não remove as tabelas, visões ou outros objetos pertencentes ao usuário. Se o usuário possuir algum banco de dados uma mensagem de erro é gerada.

## Parâmetros

*nome*

O nome do usuário a ser removido.

## Observações

O PostgreSQL inclui o aplicativo *dropuser* que possui a mesma funcionalidade deste comando (na verdade, chama este comando), mas que pode ser executada a partir da linha de comandos.

Para remover um usuário que possui um banco de dados, primeiro o banco de dados deve ser removido ou mudado de dono.

Não se aconselha remover um usuário que possua algum objeto de banco de dados, ou que tenha permissões concedidas para objetos. Atualmente isto só é verificado no caso dos donos de bancos de dados, mas é provável que as versões futuras do PostgreSQL verifiquem os outros casos.

## Exemplos

Para remover uma conta de usuário:

```
DROP USER josias;
```

## Compatibilidade

O comando `DROP USER` é uma extensão do PostgreSQL. O padrão SQL deixa a definição de usuários para a implementação

## Consulte também

*ALTER USER*, *CREATE USER*

# DROP VIEW

## Nome

DROP VIEW — remove uma visão

## Sinopse

```
DROP VIEW nome [ , ... ] [ CASCADE | RESTRICT ]
```

## Descrição

O comando DROP VIEW remove uma visão existente. Para executar este comando é necessário ser o dono da visão.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) da visão a ser removida.

CASCADE

Remove automaticamente os objetos que dependem da visão (como outras visões).

RESTRICT

Não permite remover a visão caso existam objetos que dependam da mesma. Este é o padrão.

## Exemplos

Este comando remove a visão chamada `tipos`:

```
DROP VIEW tipos;
```

## Compatibilidade

Este comando está em conformidade com o padrão SQL, exceto que o padrão permite a remoção de apenas uma visão por comando.

## Consulte também

*CREATE VIEW*

# END

## Nome

END — efetiva a transação corrente

## Sinopse

```
END [ WORK | TRANSACTION ]
```

## Descrição

O comando `END` efetiva a transação corrente. Todas as modificações efetuadas pela transação se tornam visíveis para os outros, e existe a garantia de permanecerem se uma falha ocorrer. Este comando é uma extensão do PostgreSQL equivalente ao *COMMIT*.

## Parâmetros

WORK

TRANSACTION

Palavras chave opcionais. Não produzem nenhum efeito.

## Observações

Use o *ROLLBACK* para interromper a transação.

A utilização do `END` fora de uma transação não causa nenhum problema, mas provoca uma mensagem de advertência.

## Exemplos

Para efetivar a transação corrente e tornar todas as modificações permanentes:

```
END ;
```

## Compatibilidade

O comando `END` é uma extensão do PostgreSQL que fornece uma funcionalidade equivalente ao *COMMIT*, que é especificado no padrão SQL.

## Consulte também

*BEGIN*, *COMMIT*, *ROLLBACK*

# EXECUTE

## Nome

EXECUTE — executa um comando preparado

## Sinopse

```
EXECUTE nome_do_plano [ (parâmetro [, ...] ) ]
```

## Descrição

O comando `EXECUTE` é utilizado para executar um comando previamente preparado. Como os comandos preparados existem somente durante o período da sessão, o comando preparado deve ter sido criado por um comando `PREPARE` executado anteriormente na sessão corrente.

Se o comando `PREPARE` que criou o comando especificar alguns parâmetros, um conjunto compatível de parâmetros deve ser passado para o comando `EXECUTE`, senão um erro será gerado. Deve ser observado que (diferentemente das funções) os comandos preparados não são sobrecarregados baseado no tipo ou número de seus parâmetros: o nome de um comando preparado deve ser único para a sessão.

Para obter informações adicionais sobre a criação e utilização de comandos preparados consulte o comando `PREPARE`.

## Parâmetros

*nome\_do\_plano*

O nome do comando preparado a ser executado.

*parâmetro*

O valor para o parâmetro do comando preparado. Deve ser uma expressão que produz um valor com tipo de dado compatível com o tipo de dado especificado para a posição deste parâmetro, no comando `PREPARE` que criou o comando preparado.

## Saídas

A mensagem retornada pelo comando `EXECUTE` é a mensagem do comando preparado, e não do `EXECUTE`.

## Exemplos

São mostrados exemplos na seção *Exemplos* da documentação do comando `PREPARE`.

## Compatibilidade

O padrão SQL inclui o comando `EXECUTE`, mas somente para ser utilizado na linguagem SQL incorporada (embedded). Esta versão do comando `EXECUTE` também utiliza uma sintaxe um pouco diferente.

## Consulte também

`DEALLOCATE`, `PREPARE`

# EXPLAIN

## Nome

EXPLAIN — mostra o plano de execução de um comando

## Sinopse

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] comando
```

## Descrição

Este comando mostra o plano de execução que o planejador do PostgreSQL gera para o comando fornecido. O plano de execução mostra como as tabelas referenciadas pelo comando são varridas — por uma varredura sequencial simples, varredura pelo índice, etc. — e, se várias tabelas forem referenciadas, quais algoritmos de junção são utilizados para unir as linhas das tabelas de entrada.

Do que é mostrado, a parte mais importante é o custo estimado de execução do comando, que é a estimativa feita pelo planejador de quanto tempo demora para executar o comando (medido em unidades de acesso às páginas do disco). Na verdade, são mostrados dois números: o tempo inicial antes que a primeira linha possa ser retornada, e o tempo total para retornar todas as linhas. Para a maior parte dos comandos o tempo total é o que importa, mas em contextos como uma subseleção no `EXISTS`, o planejador escolhe o menor tempo inicial em vez do menor tempo total (porque o executor pára após ter obtido uma linha). Além disso, se for limitado o número de linhas retornadas usando a cláusula `LIMIT`, o planejador efetua uma interpolação apropriada entre estes custos para saber qual é realmente o plano de menor custo.

A opção `ANALYZE` faz o comando ser realmente executado, e não apenas planejado. O tempo total decorrido gasto em cada nó do plano (em milissegundos) e o número total de linhas realmente retornadas são adicionados ao que é mostrado. Esta opção é útil para ver se as estimativas do planejador estão próximas da realidade.

**Importante:** Deve-se ter em mente que o comando é realmente executado quando a opção `ANALYZE` é utilizada. Embora o comando `EXPLAIN` despreze qualquer saída produzida pelo `SELECT`, os outros efeitos colaterais do comando ocorrem da forma usual. Se for desejado utilizar `EXPLAIN ANALYZE` em um comando `INSERT`, `UPDATE`, `DELETE` ou `EXECUTE` sem afetar os dados, utilize o seguinte procedimento:

```
BEGIN;  
EXPLAIN ANALYZE ...;  
ROLLBACK;
```

## Parâmetros

`ANALYZE`

Executar o comando e mostrar os tempos reais de execução.

`VERBOSE`

Mostrar a representação interna completa da árvore do plano, em vez de apenas um resumo. Geralmente esta opção é útil apenas para finalidades especiais de depuração. A saída produzida pela opção `VERBOSE` é formatada para ser facilmente lida (`pretty-print`) ou não, dependendo de como estiver definido o parâmetro de configuração `explain_pretty_print`.

*comando*

Qualquer comando `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `EXECUTE` ou `DECLARE`, cujo plano de execução se deseja ver.

## Observações

Existe apenas documentação esparsa sobre o uso da informação do custo do otimizador no PostgreSQL. Consulte a Seção 13.1 para obter informações adicionais.

Para que o planejador de comandos do PostgreSQL esteja razoavelmente informado para tomar decisões ao otimizar os comandos, o comando `ANALYZE` deve ser executado para registrar as estatísticas sobre a distribuição dos dados dentro da tabela. Se isto não tiver sido feito (ou se a distribuição estatística dos dados da tabela mudou de forma significativa desde a última vez que o comando `ANALYZE` foi executado), os custos estimados têm pouca chance de estarem em conformidade com as verdadeiras propriedades do comando e, conseqüentemente, pode ser escolhido um plano de comando inferior.

Antes do PostgreSQL 7.3, o plano era mostrado na forma de uma mensagem `NOTICE`. Agora aparece na forma do resultado de uma consulta (formatado como uma tabela de uma única coluna do tipo texto).

## Exemplos

Mostrar o plano para uma consulta simples em uma tabela com uma única coluna `integer` e 10.000 linhas:

```
EXPLAIN SELECT * FROM foo;
```

```

              QUERY PLAN
-----
Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 linha)
```

Havendo um índice, e sendo feita uma consulta com uma condição `WHERE` indexável, o comando `EXPLAIN` pode mostrar um plano diferente:

```
EXPLAIN SELECT * FROM foo WHERE i = 4;
```

```

              QUERY PLAN
-----
Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 linhas)
```

O exemplo abaixo mostra o plano para uma consulta contendo uma função de agregação:

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;
```

```

              QUERY PLAN
-----
Aggregate  (cost=23.93..23.93 rows=1 width=4)
->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6 width=4)
     Index Cond: (i < 10)
(3 linhas)
```

Abaixo está um exemplo da utilização do comando `EXPLAIN EXECUTE` para mostrar o plano para uma consulta preparada:

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
    WHERE id > $1 AND id < $2
    GROUP BY foo;
```

```
EXPLAIN ANALYZE EXECUTE query(100, 200);
```

```

              QUERY PLAN
-----
HashAggregate  (cost=39.53..39.53 rows=1 width=8) (actual time=0.661..0.672 rows=7 loops=1)
->  Index Scan using test_pkey on test  (cost=0.00..32.97 rows=1311 width=8) (actual
time=0.050..0.395 rows=99 loops=1)
     Index Cond: ((id > $1) AND (id < $2))
Total runtime: 0.851 ms
(4 linhas)
```

Obviamente, os números específicos mostrados aqui dependem do conteúdo real das tabelas envolvidas. Deve ser observado, também, que os números, e mesmo a estratégia selecionada para o comando, podem variar entre versões diferentes do PostgreSQL devido a melhorias no planejador. Além disso, o comando `ANALYZE` utiliza amostragem aleatória



para estimar as estatísticas dos dados; portanto, é possível que as estimativas de custo mudem após a execução do comando *ANALYZE*, mesmo que a distribuição real dos dados da tabela não tenha mudado.

## **Compatibilidade**

Não existe o comando *EXPLAIN* no padrão SQL.

## **Consulte também**

*ANALYZE*

# FETCH

## Nome

FETCH — traz linhas de uma consulta usando um cursor

## Sinopse

```
FETCH [ direção { FROM | IN } ] nome_do_cursor
```

onde *direção*

pode ser omitida ou pode ser um dentre:

```
NEXT
PRIOR
FIRST
LAST
ABSOLUTE contador
RELATIVE contador
contador
ALL
FORWARD
FORWARD contador
FORWARD ALL
BACKWARD
BACKWARD contador
BACKWARD ALL
```

## Descrição

O comando `FETCH` traz linhas utilizando um cursor previamente criado.

O cursor possui uma posição associada, a qual é utilizada pelo comando `FETCH`. A posição do cursor pode ser: antes da primeira linha, em uma determinada linha, ou após a última linha do resultado da consulta. Ao ser criado, o cursor fica posicionado antes da primeira linha. Após trazer algumas linhas, o cursor fica posicionado na linha trazida mais recentemente. Se o comando `FETCH` ultrapassar o final das linhas disponíveis, então o cursor fica posicionado após a última linha, ou antes da primeira linha se estiver trazendo para trás. Os comandos `FETCH ALL` e `FETCH BACKWARD ALL` sempre deixam o cursor posicionado após a última linha ou antes da primeira linha, respectivamente.

As formas `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE` e `RELATIVE` trazem uma única linha após mover o cursor de forma apropriada. Se a linha não existir, um resultado vazio é retornado, e o cursor é deixado posicionado antes da primeira linha ou após a última linha, conforme seja apropriado.

As formas que utilizam `FORWARD` e `BACKWARD` trazem o número indicado de linhas movendo o cursor para frente ou para trás, deixando o cursor posicionado na última linha retornada; ou após/antes de todas as linhas se o *contador* exceder o número de linhas disponíveis.

`RELATIVE 0`, `FORWARD 0` e `BACKWARD 0` requerem que seja trazida a linha corrente sem mover o cursor, ou seja, traz novamente a linha trazida mais recentemente. Sempre é bem-sucedido, a menos que o cursor esteja posicionado antes da primeira linha ou após a última linha; nestes casos, nenhuma linha é retornada.

## Parâmetros

*direção*

Define a *direção* para trazer e o número de linhas a serem trazidas. Pode ser um entre os seguintes:

`NEXT`

Traz a próxima linha. Este é o padrão se a *direção* for omitida.

PRIOR

Traz a linha anterior.

FIRST

Traz a primeira linha da consulta (o mesmo que ABSOLUTE 1).

LAST

Traz a última linha da consulta (o mesmo que ABSOLUTE -1).

ABSOLUTE *contador*

Traz a *contador*-ésima linha da consulta, ou a  $\text{abs}(\text{contador})$ -ésima linha a partir do fim se *contador* for negativo. Posiciona antes da primeira linha ou após a última linha se o *contador* estiver fora do intervalo; em particular, ABSOLUTE 0 posiciona antes da primeira linha.

RELATIVE *contador*

Traz a *contador*-ésima linha à frente, ou a  $\text{abs}(\text{contador})$ -ésima linha atrás se o *contador* for negativo. RELATIVE 0 traz novamente a linha corrente, se houver.

*contador*

Traz as próximas *contador* linhas (o mesmo que FORWARD *contador*).

ALL

Traz todas as linhas restantes (o mesmo que FORWARD ALL).

FORWARD

Traz a próxima linha (o mesmo que NEXT).

FORWARD *contador*

Traz as próximas *contador* linhas. FORWARD 0 traz novamente a linha corrente.

FORWARD ALL

Traz todas as linhas restantes.

BACKWARD

Traz a linha anterior (o mesmo que PRIOR).

BACKWARD *contador*

Traz as *contador* linhas anteriores (varrendo para trás). BACKWARD 0 traz novamente a linha corrente.

BACKWARD ALL

Traz todas as linhas anteriores (varrendo para trás).

*contador*

O *contador* é uma constante inteira, possivelmente com sinal, que determina a posição ou o número de linhas a serem trazidas. Para os casos FORWARD e BACKWARD, especificar um *contador* negativo é equivalente a mudar o sentido de FORWARD e BACKWARD.

*nome\_do\_cursor*

O nome de um cursor aberto.

## Saídas

Ao terminar bem-sucedido, o comando FETCH retorna uma linha de fim de comando na forma

FETCH *contador*

O *contador* é o número de linhas trazidas (possivelmente zero). Deve ser observado que no psql a linha de fim de comando não é exibida, uma vez que o psql mostra as linhas trazidas em seu lugar.

## Observações

O cursor deve ser declarado com a opção `SCROLL` se houver intenção de utilizar qualquer variante do comando `FETCH` que não seja `FETCH NEXT` ou `FETCH FORWARD` com um contador positivo. Para consultas simples o PostgreSQL permite trazer para trás usando cursores não declarados com `SCROLL`, mas é bom não confiar neste comportamento. Se o cursor for declarado com `NO SCROLL`, então trazer para trás não é permitido.

Trazer com `ABSOLUTE` não é nem um pouco mais rápido que navegar para a linha desejada usando um movimento relativo: a implementação subjacente necessita de qualquer maneira percorrer todas as linhas intermediárias. Buscas absolutas negativas são piores ainda: a consulta precisa ser lida até o fim para encontrar a última linha, e depois percorrida para trás a partir deste ponto. Entretanto, voltar para o início da consulta (como com `FETCH ABSOLUTE 0`) é rápido.

A atualização dos dados através de cursores não é permitida atualmente pelo PostgreSQL.

O comando `DECLARE` é utilizado para definir o cursor. Deve ser utilizado o comando `MOVE` para mudar a posição do cursor sem trazer dados.

## Exemplos

O exemplo a seguir percorre uma tabela usando um cursor.

```
BEGIN WORK;
```

```
-- Definir o cursor:
```

```
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM filmes;
```

```
-- Trazer as 5 primeiras linhas do cursor liahona:
```

```
FETCH FORWARD 5 FROM liahona;
```

cod	titulo	did	data_prod	tipo	duracao
BL101	The Third Man	101	1949-12-23	Drama	01:44
BL102	The African Queen	101	1951-08-11	Romance	01:43
JL201	Une Femme est une Femme	102	1961-03-12	Romance	01:25
P_301	Vertigo	103	1958-11-14	Ação	02:08
P_302	Becket	103	1964-02-03	Drama	02:28

```
-- Trazer a linha anterior:
```

```
FETCH PRIOR FROM liahona;
```

cod	titulo	did	data_prod	tipo	duracao
P_301	Vertigo	103	1958-11-14	Ação	02:08

```
-- Fechar o cursor e terminar a transação:
```

```
CLOSE liahona;
```

```
COMMIT WORK;
```

## Compatibilidade

O padrão SQL define o comando `FETCH` apenas para uso na linguagem SQL incorporada (embedded). Esta variante do `FETCH` descrita aqui retorna os dados como se fossem o resultado de um comando `SELECT`, em vez de colocar nas variáveis do hospedeiro. Fora este ponto, o comando `FETCH` possui total compatibilidade ascendente<sup>1</sup> com o padrão SQL.

As formas do comando `FETCH` envolvendo `FORWARD` e `BACKWARD`, assim bem como todas as formas de `FETCH contador` e `FETCH ALL`, na qual o `FORWARD` está implícito,

O padrão SQL permite apenas o `FROM` precedendo o nome do cursor; a opção para utilizar `IN` é uma extensão.

## Consulte também

`CLOSE`, `DECLARE`, `MOVE`

## Notas

1. upward compatibility — (compatibilidade ascendente) característica de um software que funciona sem modificações em versões mais recentes ou mais avançadas de determinado sistema de computador. Webster's New World Dicionário de Informática, Bryan Pfaffenberger, Editora Campus, 1999. (N. do T.)

# GRANT

## Nome

GRANT — define privilégios de acesso

## Sinopse

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER }
        [,...] | ALL [ PRIVILEGES ] }
ON [ TABLE ] nome_da_tabela [, ...]
TO { nome_do_usuario | GROUP nome_do_grupo | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
ON DATABASE nome_do_banco_de_dados [, ...]
TO { nome_do_usuario | GROUP nome_do_grupo | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
ON FUNCTION nome_da_função ([tipo, ...]) [, ...]
TO { nome_do_usuario | GROUP nome_do_grupo | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
ON LANGUAGE nome_da_linguagem [, ...]
TO { nome_do_usuario | GROUP nome_do_grupo | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
ON SCHEMA nome_do_esquema [, ...]
TO { nome_do_usuario | GROUP nome_do_grupo | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
ON TABLESPACE nome_do_espaco_de_tabelas [, ...]
TO { nome_do_usuario | GROUP nome_do_grupo | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

## Descrição

O comando `GRANT` concede privilégios específicos para um objeto (tabela, visão, seqüência, banco de dados, função, linguagem procedural, esquema ou espaço de tabelas) para um ou mais usuários ou grupos de usuários. Estes privilégios são adicionados aos já concedidos, caso existam.

A palavra chave `PUBLIC` indica que o privilégio deve ser concedido para todos os usuários, inclusive aos que vierem a ser criados posteriormente. `PUBLIC` pode ser considerado como um grupo definido implicitamente que sempre inclui todos os usuários. Um determinado usuário possui a soma dos privilégios concedidos diretamente para o mesmo, mais os privilégios concedidos para todos os grupos que este seja membro, mais os privilégios concedidos para `PUBLIC`.

Se for especificado `WITH GRANT OPTION` quem recebe o privilégio pode, por sua vez, conceder o privilégio a outros. Sem esta opção de concessão quem recebe não pode conceder o privilégio. Atualmente as opções de concessão somente podem ser concedidas a usuários individuais, e não a grupos ou `PUBLIC`.

Não é necessário conceder privilégios para o dono do objeto (geralmente o usuário que o criou), porque o dono possui todos os privilégios por padrão (Entretanto, o dono pode decidir revogar alguns de seus próprios privilégios por motivo de segurança). O direito de remover um objeto, ou de alterar a sua definição de alguma forma, não é descrito por um privilégio que possa ser concedido; é inerente ao dono e não pode ser concedido ou revogado. O dono possui também, implicitamente, todas as opções de concessão para o objeto.

Dependendo do tipo do objeto, os privilégios iniciais padrão podem incluir a concessão de alguns privilégios para `PUBLIC`. O padrão é: não permitir o acesso público para tabelas e esquemas; privilégio de criação de tabela `TEMP` para bancos de dados; privilégio `EXECUTE` para funções; e privilégio `USAGE` para linguagens. O dono do objeto pode, é claro, revogar estes privilégios (para a máxima segurança deve ser executado o comando `REVOKE` na mesma transação que criar o objeto; dessa forma não haverá tempo para que outro usuário possa usar o objeto).

Os privilégios possíveis são:

#### SELECT

Permite consultar (*SELECT*) qualquer coluna da tabela, visão ou seqüência especificada. Também permite utilizar o comando *COPY TO*. Para as seqüências, este privilégio também permite o uso da função *currval*.

#### INSERT

Permite inserir (*INSERT*) novas linhas na tabela especificada. Também permite utilizar o comando *COPY FROM*.

#### UPDATE

Permite modificar (*UPDATE*) os dados de qualquer coluna da tabela especificada. O comando *SELECT ... FOR UPDATE* também requer este privilégio (além do privilégio *SELECT*). Para as seqüências, este privilégio permite o uso das funções *nextval* e *setval*.

#### DELETE

Permite excluir (*DELETE*) linhas da tabela especificada.

#### RULE

Permite criar regras para a tabela ou para a visão (Consulte o comando *CREATE RULE*).

#### REFERENCES

Para criar uma restrição de chave estrangeira é necessário possuir este privilégio, tanto na tabela que faz referência quanto na tabela que é referenciada.

#### TRIGGER

Permite criar gatilhos na tabela especificada (Consulte o comando *CREATE TRIGGER*).

#### CREATE

Para bancos de dados, permite a criação de novos esquemas no banco de dados.

Para esquemas, permite a criação de novos objetos no esquema. Para mudar o nome de um objeto existente é necessário ser o dono do objeto e possuir este privilégio no esquema que o contém.

Para espaços de tabelas, permite a criação de tabelas e índices no espaço de tabelas, e permite também a criação de bancos de dados que possuam este espaço de tabelas como seu espaço de tabelas padrão (Deve ser observado que revogar este privilégio não altera a colocação dos objetos existentes).

#### TEMPORARY

#### TEMP

Permite a criação de tabelas temporárias ao usar o banco de dados.

#### EXECUTE

Permite utilizar a função especificada e qualquer operador implementado por cima da função. Este é o único tipo de privilégio aplicável às funções (Esta sintaxe funciona para as funções de agregação, da mesma forma).

#### USAGE

Para as linguagens procedurais, permite o uso da linguagem especificada para criar funções nesta linguagem. Este é o único tipo de privilégio aplicável às linguagens procedurais.

Para os esquemas, permite acessar os objetos contidos no esquema especificado (assumindo que os privilégios requeridos para os próprios objetos estejam atendidos). Essencialmente, permite a quem recebe a concessão “procurar” por objetos dentro do esquema.

#### ALL PRIVILEGES

Concede todos os privilégios disponíveis de uma só vez. A palavra chave *PRIVILEGES* é opcional no PostgreSQL, embora seja requerida pelo SQL estrito.

Os privilégios requeridos por outros comandos estão listados nas páginas de referência dos respectivos comandos.

## Observações

O comando *REVOKE* é utilizado para revogar privilégios de acesso.

Quando alguém, que não é dono do objeto, tenta conceder privilégios para o objeto, o comando falha imediatamente se o usuário não possuir algum privilégio para o objeto. Desde que algum privilégio esteja disponível o comando prossegue, mas só concede os privilégios para os quais o usuário tem a opção de concessão. A forma *GRANT ALL PRIVILEGES* emite uma mensagem de advertência se nenhuma opção de concessão for possuída, enquanto as outras formas emitem mensagem de advertência se a opção de concessão para algum dos privilégios especificamente nomeados no comando não for possuída (Em princípio estas declarações também se aplicam ao dono do objeto, mas como o dono é sempre tratado como possuindo todas as opções de concessão estes casos nunca ocorrem).

Deve ser observado que os superusuários do banco de dados podem acessar todos os objetos, sem considerar os privilégios definidos para o objeto. Isto é comparável aos direitos do usuário *root* no sistema operacional Unix. Assim como no caso do *root*, não é aconselhável operar como um superusuário a não ser quando for absolutamente necessário.

Se um superusuário decidir executar o comando *GRANT* ou o comando *REVOKE*, o comando é executado como se tivesse sido executado pelo dono do objeto afetado. Em particular, os privilégios concedidos através deste comando aparecem como se tivessem sido concedidos pelo dono do objeto.

Atualmente o PostgreSQL não suporta conceder ou revogar privilégios para colunas individuais da tabela. Uma forma possível de transpor esta limitação é criar uma visão possuindo apenas as colunas desejadas e, então, conceder os privilégios para a visão.

Pode ser usado o comando *\z* do aplicativo *psql* para obter informações sobre os privilégios concedidos como, por exemplo:

```
=> \z minha_tabela
```

```

                Access privileges for database "lusitania"
Schema |      Name      | Type |      Access privileges
-----+-----+-----+-----
public | minha_tabela | table | {miriam=arwdRxt/miriam,=r/miriam,"group todos=arw/miriam"}
(1 linha)
```

As entradas mostradas pelo comando *\z* são interpretadas da seguinte forma:

```

=xxxx -- privilégios concedidos para PUBLIC
uname=xxxx -- privilégios concedidos para o usuário
group gname=xxxx -- privilégios concedidos para o grupo

r -- SELECT ("read")
w -- UPDATE ("write")
a -- INSERT ("append")
d -- DELETE
R -- RULE
x -- REFERENCES
t -- TRIGGER
X -- EXECUTE
U -- USAGE
C -- CREATE
T -- TEMPORARY
arwdRxt -- ALL PRIVILEGES (para tabelas)
* -- opção de concessão para o privilégio precedente

/yyyy -- usuário que concedeu este privilégio
```

O exemplo mostrado acima seria visto pela usuária *miriam* após esta ter criado a tabela *minha\_tabela* e executado:

```
GRANT SELECT ON minha_tabela TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON minha_tabela TO GROUP todos;
```



Se a coluna “Access privileges” estiver vazia para um determinado objeto, isto significa que o objeto possui os privilégios padrão (ou seja, suas colunas de privilégio são nulas). Os privilégios padrão sempre incluem todos os privilégios para o dono, e podem incluir alguns privilégios para `PUBLIC` dependendo do tipo do objeto, como foi explicado acima. O primeiro comando `GRANT` ou `REVOKE` em um objeto cria uma instância dos privilégios padrão (produzindo, por exemplo, `{=,miriam=arwdRxt}`) e, em seguida, modifica esta instância de acordo com a solicitação especificada.

Deve ser observado que as opções de concessão implícitas do dono não são marcadas na visualização dos privilégios de acesso. O `*` aparece somente quando as opções de concessão foram concedidas explicitamente para alguém.

## Exemplos

Conceder, para todos os usuários, o privilégio de inserção na tabela `filmes`:

```
GRANT INSERT ON filmes TO PUBLIC;
```

Conceder para o usuário `manuel` todos os privilégios disponíveis na visão `tipos`:

```
GRANT ALL PRIVILEGES ON tipos TO manuel;
```

Deve ser observado que, embora o comando acima realmente concede todos os privilégios se for executado por um superusuário ou pelo dono de `tipos`, se for executado por outra pessoa somente concederá as permissões para as quais esta outra pessoa tiver a opção de conceder.

## Compatibilidade

De acordo com o padrão SQL, a palavra chave `PRIVILEGES` em `ALL PRIVILEGES` é requerida. O padrão SQL não permite definir privilégios para mais de um objeto por comando.

O PostgreSQL permite ao dono do objeto revogar seus próprios privilégios ordinários: por exemplo, o dono da tabela pode tornar a tabela somente de leitura para ele mesmo revogando seus próprios privilégios de `INSERT`, `UPDATE` e `DELETE`. De acordo com o padrão SQL isto não é possível. A razão é que o PostgreSQL trata os privilégios do dono como tendo sido concedidos pelo dono para ele mesmo; portanto ele também pode revogá-los. No padrão SQL os privilégios do dono são concedidos pela entidade assumida “`_SYSTEM`”. Em não sendo o “`_SYSTEM`”, o dono não pode revogar estes direitos.

O padrão SQL permite definir privilégios para as colunas individuais da tabela:

```
GRANT privilégios
    ON tabela [ ( coluna [, ...] ) ] [, ...]
    TO { PUBLIC | nome_do_usuario [, ...] } [ WITH GRANT OPTION ]
```

O padrão SQL permite conceder o privilégio `USAGE` em outros tipos de objeto: conjuntos de caracteres, classificações (*collations*<sup>1</sup>), traduções e domínios.

O privilégio `RULE`, e os privilégios para bancos de dados, espaços de tabelas, esquemas, linguagens e seqüências são extensões do PostgreSQL.

## Consulte também

*REVOKE*

## Notas

1. *collation*; *collating sequence* — Um método para comparar duas cadeias de caracteres comparáveis. Todo conjunto de caracteres possui seu *collation* padrão. (Second Informal Review Draft) ISO/IEC 9075:1992, Database Language SQL- July 30, 1992. (N. do T.)

# INSERT

## Nome

INSERT — cria novas linhas na tabela

## Sinopse

```
INSERT INTO tabela [ ( coluna [, ...] ) ]  
    { DEFAULT VALUES | VALUES ( { expressão | DEFAULT } [, ...] ) | consulta }
```

## Descrição

O comando `INSERT` permite inserir novas linhas na tabela. Pode ser inserida uma única linha especificada por expressões de valor, ou várias linhas resultantes de uma consulta.

Os nomes das colunas de destino podem ser listados em qualquer ordem. Se não for fornecida nenhuma lista de nomes de colunas, o padrão é usar todas as colunas da tabela na ordem em que foram declaradas; ou os primeiros *N* nomes de colunas, se existirem apenas *N* colunas fornecidas na cláusula `VALUES` ou na *consulta*. Os valores fornecidos pela cláusula `VALUES` e pela *consulta* são associados à lista de colunas explícita ou implícita da esquerda para a direita.

As colunas que não estão presentes na lista de colunas explícita ou implícita são preenchidas com o valor padrão, seja o valor padrão declarado ou nulo se não houver nenhum.

Se a expressão para alguma coluna não for do tipo de dado correto, será tentada uma conversão automática de tipo.

É necessário possuir o privilégio `INSERT` na tabela para poder inserir linhas. Se for utilizada a cláusula *consulta* para inserir linhas a partir de uma consulta, também é necessário possuir o privilégio `SELECT` em todas as tabelas usadas pela consulta.

## Parâmetros

*tabela*

O nome (opcionalmente qualificado pelo esquema) de uma tabela existente.

*coluna*

O nome de uma coluna da *tabela*. O nome da coluna pode ser qualificado por um nome de subcampo ou por um índice de matriz, se for necessário (a inserção em apenas alguns campos de uma coluna composta deixa os outros campos nulos).

`DEFAULT VALUES`

Todas as colunas são preenchidas com seu valor padrão.

*expressão*

Uma expressão ou valor a ser atribuído à *coluna* correspondente.

`DEFAULT`

A *coluna* correspondente é preenchida com o valor padrão.

*consulta*

Uma consulta (comando `SELECT`) que fornece as linhas a serem inseridas. Consulte o comando `SELECT` para obter a descrição da sintaxe.

## Saídas

Ao terminar bem-sucedido, o comando `INSERT` retorna uma linha de fim de comando na forma

```
INSERT oid contador
```

O *contador* é o número de linhas inseridas. Se *contador* for igual a um, e a tabela de destino possuir OIDs, então *oid* é o OID atribuído à linha inserida, senão *oid* é zero.

## Exemplos

Inserir uma única linha na tabela `filmes`:

```
INSERT INTO filmes VALUES
  ('UA502', 'Bananas', 105, '1971-07-13', 'Comédia', '82 minutes');
```

No exemplo abaixo, a coluna `duracao` é omitida e, portanto, receberá o valor padrão:

```
INSERT INTO filmes (cod, titulo, did, data_prod, tipo)
  VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

O exemplo abaixo utiliza a cláusula `DEFAULT` para as colunas de data em vez de especificar um valor.

```
INSERT INTO filmes VALUES
  ('UA502', 'Bananas', 105, DEFAULT, 'Comédia', '82 minutes');
INSERT INTO filmes (cod, titulo, did, data_prod, tipo)
  VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

Para inserir uma linha consistindo inteiramente de valores padrão:

```
INSERT INTO filmes DEFAULT VALUES;
```

O exemplo abaixo insere algumas linhas na tabela `filmes` a partir da tabela `temp_filmes` com a mesma disposição de colunas da tabela `filmes`:

```
INSERT INTO filmes SELECT * FROM temp_filmes WHERE data_prod < '2004-05-07';
```

O exemplo mostrado abaixo insere em colunas de matriz:

```
-- Criar um tabuleiro vazio de 3x3 posições para o Jogo da Velha
-- (estes comandos criam o mesmo tabuleiro)
INSERT INTO tictactoe (game, board[1:3][1:3])
  VALUES (1, '{{"", "", ""}, {"", "", ""}, {"", "", ""}}');
INSERT INTO tictactoe (game, board)
  VALUES (2, '{{{,}, {,}, {,}}, {{,}, {{,}, {{,}}}');
```

## Compatibilidade

O comando `INSERT` está em conformidade com o padrão SQL. O caso em que a lista de nomes de colunas é omitida, mas nem todas as colunas são preenchidas a partir da cláusula `VALUES` ou da *consulta* não é permitido pelo padrão.

As possíveis limitações da cláusula *consulta* estão documentadas no comando `SELECT`.

# LISTEN

## Nome

LISTEN — ouve uma notificação

## Sinopse

LISTEN *nome*

## Descrição

O comando LISTEN registra a sessão corrente como ouvinte da condição de notificação *nome*. Se a sessão corrente já estiver registrada como ouvinte desta condição de notificação, nada é feito.

Sempre que o comando NOTIFY *nome* é executado, tanto por esta sessão quanto por outra conectada ao mesmo banco de dados, todas as sessões ouvindo esta condição de notificação no momento são notificadas, e cada uma por sua vez notifica seu aplicativo cliente conectado. Consulte a documentação do comando NOTIFY para obter informações adicionais.

A sessão pode cancelar o registro para uma determinada condição de notificação utilizando o comando UNLISTEN. Os registros da sessão como ouvinte são automaticamente removidos quando a sessão termina.

O método que o aplicativo cliente deve usar para detectar eventos de notificação depende da interface de programação de aplicativos (API) do PostgreSQL que o aplicativo usa. Quando usa a biblioteca libpq o aplicativo executa o comando LISTEN como um comando SQL comum, e depois deve chamar periodicamente a função PQnotifies para descobrir se algum evento de notificação foi recebido. Outras interfaces, como a libpqctl, fornecem métodos de mais alto nível para tratar os eventos de notificação; na verdade, usando a libpqctl o programador do aplicativo não deve nem mesmo executar o comando LISTEN ou UNLISTEN diretamente. Consulte a documentação da interface sendo usada para obter mais detalhes.

O comando NOTIFY contém uma explicação mais extensa sobre a utilização do comando LISTEN e do comando NOTIFY.

## Parâmetros

*nome*

O nome da condição de notificação (qualquer identificador).

## Exemplos

Configurar e executar a sequência ouvir/notificar a partir do psql:

```
LISTEN virtual;  
NOTIFY virtual;  
Asynchronous notification "virtual" received from server process with PID 8448.
```

## Compatibilidade

Não existe o comando LISTEN no padrão SQL.

## Consulte também

NOTIFY, UNLISTEN

# LOAD

## Nome

LOAD — carrega ou recarrega um arquivo de biblioteca compartilhada

## Sinopse

```
LOAD 'nome_do_arquivo'
```

## Descrição

Este comando carrega um arquivo de biblioteca compartilhada no espaço de endereçamento do servidor PostgreSQL. Se o arquivo tiver sido carregado anteriormente, primeiro é descarregado. Este comando é útil, principalmente, para descarregar e recarregar um arquivo de biblioteca compartilhada modificado após ter sido carregado pelo servidor. Para usar a biblioteca compartilhada, as funções na mesma devem ser declaradas usando o comando *CREATE FUNCTION*.

O nome do arquivo é especificado da mesma forma que os nomes de bibliotecas compartilhadas no comando *CREATE FUNCTION*; em particular, pode-se confiar no caminho de procura e na adição automática da extensão de nome de arquivo de biblioteca compartilhada padrão do sistema. Consulte a Seção 31.3 para obter informações adicionais sobre este assunto.

## Compatibilidade

O comando LOAD é uma extensão do PostgreSQL.

## Consulte também

*CREATE FUNCTION*

# LOCK

## Nome

LOCK — bloqueia uma tabela

## Sinopse

```
LOCK [ TABLE ] nome [, ...] [ IN modo_de_bloqueio MODE ] [ NOWAIT ]
```

onde *modo\_de\_bloqueio* é um entre:

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE  
| SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

## Descrição

O comando `LOCK TABLE` obtém um bloqueio no nível de tabela aguardando, quando necessário, pela liberação de qualquer bloqueio conflitante. Se for especificado `NOWAIT`, então o comando `LOCK TABLE` não fica aguardando para obter o bloqueio desejado: se não conseguir obter o bloqueio imediatamente, o comando é interrompido e um erro é emitido. Uma vez obtido, o bloqueio é mantido pelo restante da transação corrente (Não existe o comando `UNLOCK TABLE`; os bloqueios são sempre liberados no final da transação).

Ao obter automaticamente o bloqueio para os comandos que fazem referência a tabelas, o PostgreSQL sempre utiliza o modo de bloqueio menos restritivo possível. O comando `LOCK TABLE` serve para os casos onde é necessário um modo de bloqueio mais restritivo.

Por exemplo, suponha que um aplicativo executa uma transação no nível de isolamento `READ COMMITTED`, e precisa garantir que os dados da tabela permaneçam estáveis durante a transação. Para conseguir esta situação pode ser obtido o modo de bloqueio `SHARE` na tabela antes de realizar a consulta. Isto impede a alteração concorrente dos dados, garantindo que as próximas operações de leitura na tabela vão enxergar uma visão estável de dados efetivados, porque o modo de bloqueio `SHARE` conflita com o modo de bloqueio `ROW EXCLUSIVE` obtido por quem está escrevendo, fazendo com que o comando `LOCK TABLE nome IN SHARE MODE` aguarde todas as transações que obtiveram bloqueio no modo `ROW EXCLUSIVE` efetivarem ou desfazerem suas modificações. Portanto, quando este bloqueio é obtido não existem escritas não efetivadas pendentes; além disso, nenhuma transação pode começar enquanto o bloqueio não for liberado.

Para obter um efeito semelhante ao executar uma transação no nível de isolamento serializável, é necessário executar o comando `LOCK TABLE` antes de executar qualquer comando `SELECT` ou de modificação de dado. A visão dos dados de uma transação serializável é congelada no momento em que seu primeiro comando `SELECT` ou de modificação de dados começa. Um comando `LOCK TABLE` posterior na transação ainda vai impedir escritas concorrentes — mas não vai garantir que o que é lido pela transação corresponde aos últimos valores efetivados.

Se uma transação deste tipo altera os dados da tabela, então deve ser utilizado o modo de bloqueio `SHARE ROW EXCLUSIVE` em vez do modo `SHARE`, para garantir que somente uma transação deste tipo executa de cada vez. Sem isto, é possível ocorrer um impasse (`deadlock`): duas transações podem obter o modo de bloqueio `SHARE`, e depois ficarem impossibilitadas de obter o bloqueio no modo `ROW EXCLUSIVE` para realizar suas modificações (Deve ser observado que os bloqueios da própria transação nunca entram em conflito e, portanto, a transação pode obter o modo `ROW EXCLUSIVE` enquanto mantém o modo de bloqueio `SHARE` — mas não se outra transação estiver com o modo de bloqueio `SHARE`). Para evitar os impasses, deve ser garantido que as transações obtêm o bloqueio dos mesmos objetos na mesma ordem e, se vários modos de bloqueio estão envolvidos para um único objeto, as transações devem sempre obter o modo mais restritivo primeiro.

Mais informações sobre os modos de bloqueio e estratégias de bloqueio podem ser encontradas na Seção 12.3.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) da tabela existente a ser bloqueada.

O comando `LOCK TABLE a, b;` é equivalente a `LOCK TABLE a; LOCK TABLE b;`. As tabelas são bloqueadas uma a uma na ordem especificada no comando `LOCK`.

#### *modo\_de\_bloqueio*

O modo de bloqueio especifica os bloqueios com os quais este modo conflita. Os modos de bloqueio estão descritos na Seção 12.3.

Se não for especificado nenhum modo de bloqueio, então `ACCESS EXCLUSIVE`, o modo mais restritivo, é usado.

#### `NOWAIT`

Especifica que o comando `LOCK TABLE` não deve aguardar pela liberação dos bloqueios conflitantes: se os bloqueios especificados não puderem ser obtidos imediatamente, a transação é interrompida.

## Observações

O comando `LOCK TABLE ... IN ACCESS SHARE MODE` requer o privilégio `SELECT` na tabela de destino. Todas as outras formas do comando `LOCK` requerem os privilégios `UPDATE` e/ou `DELETE`.

O comando `LOCK TABLE` é útil apenas dentro de um bloco de transação (par `BEGIN/COMMIT`), uma vez que o bloqueio é liberado tão logo a transação termine. Um comando `LOCK TABLE` aparecendo fora de um bloco de transação forma uma transação auto-contida e, portanto, o bloqueio é liberado tão logo seja obtido.

O comando `LOCK TABLE` trata somente de bloqueios no nível de tabela e, portanto, os nomes dos modos contendo `ROW` são todos equivocados. Os nomes destes modos devem ser lidos geralmente como indicando a intenção de obter um bloqueio no nível de linha dentro da tabela bloqueada. Também, o modo `ROW EXCLUSIVE` é um bloqueio de tabela compartilhável. Deve-se ter em mente que todos os modos de bloqueio possuem semântica idêntica no que diz respeito ao `LOCK TABLE`, diferindo apenas nas regras sobre quais modos conflitam com quais modos. Para obter informações sobre como obter um bloqueio real no nível de linha deve ser consultada a Seção 12.3.2 e a *Cláusula FOR UPDATE* na documentação de referência do comando `SELECT`.

## Exemplos

Obter o bloqueio no modo `SHARE` da tabela que contém a chave primária, antes de fazer uma inserção na tabela que contém a chave estrangeira:

```
BEGIN WORK;
LOCK TABLE filmes IN SHARE MODE;
SELECT id FROM filmes
    WHERE nome = 'Guerra Nas Estrelas - Episódio I - A Ameaça Fantasma';
-- Executar ROLLBACK se a linha não for encontrada
INSERT INTO filmes_comentarios_usuario VALUES
    (_id_, 'Maravilhoso! Eu estava aguardando por isto há muito tempo!');
COMMIT WORK;
```

Obter o bloqueio no modo `SHARE ROW EXCLUSIVE` da tabela que contém a chave primária antes de realizar a operação de exclusão:

```
BEGIN WORK;
LOCK TABLE filmes IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM filmes_comentarios_usuario WHERE id IN
    (SELECT id FROM filmes WHERE avaliacao < 5);
DELETE FROM filmes WHERE avaliacao < 5;
COMMIT WORK;
```

## Compatibilidade

Não existe o comando `LOCK TABLE` no padrão SQL, que em seu lugar usa o comando `SET TRANSACTION` para especificar os níveis de concorrência das transações. O PostgreSQL também suporta esta funcionalidade; consulte o comando *SET TRANSACTION* para obter detalhes.

Exceto pelos modos de bloqueio `ACCESS SHARE`, `ACCESS EXCLUSIVE` e `SHARE UPDATE EXCLUSIVE`, os modos de bloqueio do PostgreSQL e a sintaxe do comando `LOCK TABLE` são compatíveis com as presentes no Oracle.

# MOVE

## Nome

MOVE — posiciona o cursor

## Sinopse

```
MOVE [ direção { FROM | IN } ] nome_do_cursor
```

## Descrição

O comando `MOVE` reposiciona o cursor sem trazer dados. O comando `MOVE` funciona exatamente como o comando `FETCH`, exceto que apenas posiciona o cursor sem retornar linhas.

Consulte o comando `FETCH` para obter detalhes sobre a sintaxe e utilização.

## Saídas

Ao terminar bem-sucedido, o comando `MOVE` retorna uma linha de fim de comando na forma

```
MOVE contador
```

O *contador* é o número de linhas que o comando `FETCH` com os mesmos parâmetros teria retornado (possivelmente zero).

## Exemplos

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM filmes;

-- Pular as primeiras 5 linhas:
MOVE FORWARD 5 IN liahona;
MOVE 5

-- Trazer a sexta linha no cursor liahona:
FETCH 1 FROM liahona;

  cod   | titulo | did | data_prod | tipo | duracao
-----+-----+-----+-----+-----+-----
P_303  | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 linha)

-- Fechar o cursor liahona e terminar a transação:
CLOSE liahona;
COMMIT WORK;
```

## Compatibilidade

Não existe o comando `MOVE` no padrão SQL.

## Consulte também

`CLOSE`, `DECLARE`, `FETCH`



# NOTIFY

## Nome

NOTIFY — gera uma notificação

## Sinopse

NOTIFY *nome*

## Descrição

O comando NOTIFY envia um evento de notificação para todos os aplicativos cliente, que tenham executado anteriormente o comando LISTEN *nome* para o nome de notificação especificado, no banco de dados corrente.

O comando NOTIFY fornece uma forma de sinal simples, ou mecanismo de comunicação entre processos, para um conjunto de processos que acessam o mesmo banco de dados do PostgreSQL. Podem ser construídos mecanismos de mais alto nível usando tabelas do banco de dados para passar dados adicionais do notificador para os ouvintes (além do mero nome da notificação).

A informação passada para o cliente por um evento de notificação inclui o nome da notificação e o PID do processo servidor da sessão notificadora. É função do projetista do banco de dados definir os nomes das notificações a serem utilizadas em um determinado banco de dados, e o significado de cada uma delas.

Usualmente, o nome da notificação é o mesmo de alguma tabela do banco de dados, e o evento de notificação essencialmente diz “Eu modifiquei esta tabela, dê uma olhada nela e veja o que há de novo”. Porém, este tipo de associação não é exigida pelos comandos NOTIFY e LISTEN. Por exemplo, um projetista de banco de dados pode usar vários nomes diferentes de notificação para sinalizar diferentes tipos de mudança em uma única tabela.

Quando o comando NOTIFY é utilizado para sinalizar a ocorrência de mudanças em uma determinada tabela, uma técnica de programação útil é colocar o comando NOTIFY na regra disparada pela atualização da tabela. Assim, a notificação acontece automaticamente quando a tabela é modificada, e o programador do aplicativo não pode acidentalmente esquecer de enviá-la.

O comando NOTIFY interage com as transações SQL de maneiras importantes. Em primeiro lugar, se o comando NOTIFY for executado dentro de uma transação o evento de notificação não será enviado até que, e a menos que, a transação seja efetivada. Este comportamento é apropriado, porque se a transação for interrompida todos os comandos dentro desta serão sem efeito, incluindo o NOTIFY. Mas, por outro lado, pode ser desconcertante quando se espera que os eventos de notificação sejam enviados imediatamente. Em segundo lugar, se a sessão ouvinte receber um sinal de notificação durante o tempo em que está executando uma transação, o evento de notificação não será entregue ao seu cliente conectado antes que a transação esteja completa (seja efetivada ou interrompida). Novamente o raciocínio é: se a notificação for enviada dentro de uma transação interrompida posteriormente, se deseja que a notificação seja desfeita de alguma maneira — mas o servidor não pode “pegar de volta” uma notificação após tê-la enviado para o cliente. Portanto, os eventos de notificação somente são entregues entre transações. O resultado final desta situação é que, os aplicativos que utilizam o comando NOTIFY para sinalização em tempo real devem tentar manter suas transações curtas.

O comando NOTIFY se comporta como os sinais do Unix sob um aspecto importante: se o mesmo nome de notificação for sinalizado diversas vezes em uma sucessão rápida, os receptores podem receber somente um evento de notificação para várias execuções do comando NOTIFY. Portanto, é uma má idéia depender do número de notificações recebidas. Em vez disso, deve ser utilizado o comando NOTIFY para acordar os aplicativos que devem prestar atenção em alguma coisa, e usado um objeto de banco de dados (como uma seqüência) para acompanhar o que aconteceu, ou quantas vezes aconteceu.

É comum a situação onde o cliente que executa o comando NOTIFY está ouvindo este nome de notificação. Neste caso, este cliente recebe o evento de notificação da mesma forma que todas as outras sessões ouvintes. Dependendo da lógica do aplicativo, pode resultar em um trabalho sem utilidade; por exemplo, ler a tabela do banco de dados para encontrar as mesmas atualizações que esta sessão acabou de fazer. É possível evitar este trabalho adicional verificando se o PID do processo servidor da sessão notificadora (presente na mensagem do evento de notificação) é o mesmo PID da própria sessão (disponível pela libpq). Quando forem idênticos, o evento de notificação é o seu próprio trabalho retornando, podendo ser ignorado (Apesar do que foi dito no parágrafo anterior, esta técnica é segura. O PostgreSQL mantém as

autonotificações separadas das notificações vindas de outras sessões e, portanto, não é possível perder uma notificação externa ignorando as próprias notificações).

## Parâmetros

*nome*

Nome da notificação a ser sinalizada (qualquer identificador).

## Exemplos

Configurar e executar a sequência ouvir/notificar usando o aplicativo psql:

```
LISTEN virtual;  
NOTIFY virtual;  
Asynchronous notification "virtual" received from server process with PID 8448.
```

## Compatibilidade

Não existe o comando NOTIFY no padrão SQL.

## Consulte também

*LISTEN, UNLISTEN*

# PREPARE

## Nome

PREPARE — prepara um comando para execução

## Sinopse

```
PREPARE nome [ (tipo_de_dado [, ...] ) ] AS comando
```

## Descrição

O comando `PREPARE` cria um comando preparado. Um comando preparado é um objeto no lado servidor que pode ser usado para otimizar o desempenho. Quando o comando `PREPARE` é executado, o comando especificado é analisado, reescrito e planejado. Após isso, quando forem emitidos comandos `EXECUTE` o comando preparado precisa apenas ser executado. Assim os estágios de análise, reescrita e planejamento são realizados apenas uma vez, e não todas as vezes que o comando é executado.

Os comandos preparados podem receber parâmetros: valores que são substituídos no comando quando este é executado. Para incluir parâmetros no comando preparado, deve ser fornecida uma lista de tipos de dado no comando `PREPARE` e no próprio comando a ser preparado, referenciando os parâmetros pela sua posição utilizando `$1`, `$2`, etc. Ao executar o comando devem ser especificados os valores reais destes parâmetros no comando `EXECUTE`. Consulte o comando `EXECUTE` para obter informações adicionais.

Os comandos preparados somente continuam existindo enquanto a sessão de banco de dados corrente existir. Quando a sessão termina o comando preparado é esquecido e, portanto, deve ser recriado antes de poder ser usado novamente. Isto significa, também, que o mesmo comando preparado não pode ser usado simultaneamente por vários clientes do banco de dados; entretanto, cada cliente pode criar e usar o seu próprio comando preparado.

A maior vantagem de desempenho dos comandos preparados acontece quando uma única sessão é usada para processar um grande número de comandos semelhantes. A diferença no desempenho é particularmente significativa quando os comandos possuem um plano ou reescrita complexos como, por exemplo, um comando envolvendo a junção de muitas tabelas, ou requerendo a aplicação de várias regras. Se o comando for relativamente simples de ser planejado e reescrito, e relativamente dispendioso para ser executado, fica mais difícil perceber a vantagem de desempenho dos comandos preparados.

## Parâmetros

*nome*

Um nome arbitrário dado a este comando preparado. Deve ser único dentro da mesma sessão, sendo usado em seguida para executar ou remover o comando preparado anteriormente.

*tipo\_de\_dado*

O tipo de dado do parâmetro do comando preparado. Para fazer referência aos parâmetros no comando preparado são usados `$1`, `$2`, etc.

*comando*

Um entre `SELECT`, `INSERT`, `UPDATE` ou `DELETE`.

## Observações

Em algumas situações o plano de comando produzido para o comando preparado será inferior ao plano de comando que seria escolhido se o comando fosse submetido e executado normalmente. Isto se deve ao fato de quando o comando é planejado, e o planejador tenta determinar o plano de comando ótimo, os valores verdadeiros dos parâmetros especificados no comando não estão disponíveis. O PostgreSQL coleta estatísticas sobre a distribuição dos dados na tabela, e pode usar valores constantes no comando para fazer suposições sobre o resultado provável da execução do comando. Como estes dados não estão disponíveis ao planejar comandos preparados com parâmetros, o plano escolhido pode ser inferior ao

ótimo. Para examinar o plano de comando escolhido pelo PostgreSQL para o comando preparado, deve ser utilizado o comando *EXPLAIN*.

Para obter informações adicionais sobre planejamento de comandos e estatísticas coletadas pelo PostgreSQL para esta finalidade, consulte a documentação do comando *ANALYZE*.

## Exemplos

Criar um comando preparado para o comando *INSERT* e, em seguida, executá-lo:

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Vale dos Caçadores', 't', 200.00);
```

Criar um comando preparado para o comando *SELECT* e, em seguida, executá-lo:

```
PREPARE usrrptplan (int, date) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

## Compatibilidade

O padrão SQL inclui o comando *PREPARE*, mas apenas para utilização na linguagem SQL incorporada (embedded). Esta versão do comando *PREPARE* também utiliza uma sintaxe um pouco diferente.

## Consulte também

*DEALLOCATE*, *EXECUTE*

# REINDEX

## Nome

REINDEX — reconstrói índices

## Sinopse

```
REINDEX { DATABASE | TABLE | INDEX } nome [ FORCE ]
```

## Descrição

O comando `REINDEX` reconstrói um índice baseado nos dados armazenados na tabela do índice, substituindo a cópia antiga do índice. Existem dois motivos principais para utilizar o comando `REINDEX`:

- O índice está corrompido, e não contém mais dados válidos. Embora na teoria esta situação nunca deva ocorrer, na prática os índices podem ficar corrompidos por causa de erros de programação ou falhas nos equipamentos. O comando `REINDEX` provê um método de recuperação.
- O índice em questão contém muitas páginas de índice mortas que não estão sendo reutilizadas. Esta situação pode acontecer com índices `B-tree` no PostgreSQL sob certos padrões de acesso. O comando `REINDEX` fornece uma maneira para reduzir o consumo de espaço do índice através da escrita de uma nova versão do índice sem as páginas mortas. Consulte a Seção 21.2 para obter informações adicionais.

## Parâmetros

**DATABASE**

Reconstrói todos os índices do sistema do banco de dados especificado. Os índices das tabelas dos usuários não são processados. Também, os índices nos catálogos do sistema compartilhados são pulados, exceto no modo autônomo (veja abaixo).

**TABLE**

Reconstrói todos os índices da tabela especificada. Se a tabela possuir uma tabela secundária “TOAST”, esta também é reindexada.

**INDEX**

Reconstrói o índice especificado.

*nome*

O nome do banco de dados, tabela ou índice a ser reindexado. Os nomes das tabelas e dos índices podem ser qualificados pelo esquema. Atualmente o comando `REINDEX DATABASE` pode reindexar apenas o banco de dados corrente e, por isso, seu parâmetro deve corresponder ao nome do banco de dados corrente.

**FORCE**

Esta é uma opção obsoleta; se for especificada será ignorada.

## Observações

Havendo suspeita que um índice de uma tabela do usuário está corrompido, é possível simplesmente reconstruir este índice, ou todos os índices da tabela, usando o comando `REINDEX INDEX` ou o comando `REINDEX TABLE`.

A situação fica mais difícil quando é necessário recuperar um índice corrompido de uma tabela do sistema. Neste caso é importante para o sistema não ter usado nenhum dos índices suspeitos (Sem dúvida, neste tipo de cenário pode acontecer que o processo servidor caia tão logo a inicialização comece por depender de índices corrompidos). Para recuperar com segurança o servidor deve ser inicializado com a opção `-P`, que impede a utilização de índice para procura em catálogo do sistema.

Uma forma de fazer, é parar o postmaster e inicializar o servidor PostgreSQL autônomo com a opção `-P` incluída na linha de comando. Em seguida pode ser executado `REINDEX DATABASE`, `REINDEX TABLE` ou `REINDEX INDEX`, dependendo de

quanto se deseja reconstruir. Em caso de dúvida deve ser usado `REINDEX DATABASE` para selecionar a reconstrução de todos os índices do sistema no banco de dados. Depois a sessão do servidor autônomo deve ser encerrada, e reiniciado o servidor normal. Consulte a página de referência do postgres para obter informações adicionais sobre como interagir com a interface do servidor autônomo.

Como alternativa, pode ser iniciada uma sessão normal do servidor com a opção `-P` incluída nas opções de linha de comando. O método para se fazer isto varia entre clientes, mas em todos os clientes baseados na biblioteca libpq é possível definir a variável de ambiente `PGOPTIONS` com o valor `-P` antes de iniciar o cliente. Deve ser observado que embora este método não requeira o bloqueio dos outros clientes, ainda assim é razoável evitar que outros usuários se conectem ao banco de dados danificado até o término dos reparos.

Havendo suspeita que algum dos índices dos catálogos do sistema compartilhados esteja corrompido (`pg_database`, `pg_group` ou `pg_shadow`), então o servidor autônomo deve ser usado para fazer o reparo. O comando `REINDEX` não processa os catálogos compartilhados no modo multiusuário.

Para todos os índices, exceto os catálogos do sistema compartilhados, o comando `REINDEX` é seguro quanto à queda e transação (`crash-safe` e `transaction-safe`). O comando `REINDEX` não é seguro quanto à queda para os índices compartilhados, e por esse motivo não é permitido durante a operação normal. Se uma falha ocorrer durante a reindexação de um destes catálogos no modo autônomo, não será possível reiniciar o servidor normal até o problema ser resolvido (O sintoma típico de um índice compartilhado reconstruído parcialmente são erros “`index is not a btree`”).

O comando `REINDEX` é semelhante a remover e recriar o índice, porque o conteúdo do índice é reconstruído a partir do início. Entretanto, as considerações sobre o bloqueio são bem diferentes. O comando `REINDEX` bloqueia a escrita mas não a leitura da tabela que está associado. Também obtém um bloqueio exclusivo do índice específico sendo processado, que bloqueia leituras que tentam utilizar o índice. Diferentemente, o comando `DROP INDEX` obtém um bloqueio exclusivo momentâneo da tabela que está associado, bloqueando tanto a escrita como a leitura. O comando `CREATE INDEX` subsequente bloqueia a escrita mas não a leitura; uma vez que o índice não está presente, nenhuma leitura vai tentar utilizá-lo, significando que não haverá bloqueios mas que as leituras podem ser forçadas a fazer varreduras sequenciais dispendiosas. Outro ponto importante é que o enfoque remover/criar invalida todos os planos de comando no `cache` que usam o índice, enquanto o comando `REINDEX` não.

Antes do PostgreSQL 7.4, o comando `REINDEX TABLE` não processava automaticamente as tabelas `TOAST` e, portanto, estas tinham de ser reindexadas através de comandos separados. Isto ainda é possível, porém redundante.

## Exemplos

Recriar os índices da tabela `minha_tabela`:

```
REINDEX TABLE minha_tabela;
```

Reconstruir um único índice:

```
REINDEX INDEX meu_indice;
```

Reconstruir todos os índices do sistema de um determinado banco de dados, sem confiar que estejam válidos:

```
$ export PGOPTIONS="-P"
$ psql bd_danificado
...
bd_danificado=> REINDEX DATABASE bd_danificado;
bd_danificado=> \q
```

## Compatibilidade

Não existe o comando `REINDEX` no padrão SQL.

# RELEASE SAVEPOINT

## Nome

RELEASE SAVEPOINT — destrói um ponto de salvamento definido anteriormente

## Sinopse

```
RELEASE [ SAVEPOINT ] ponto_de_salvamento
```

## Descrição

O comando `RELEASE SAVEPOINT` destrói o ponto de salvamento definido anteriormente na transação corrente.

Destruir um ponto de salvamento faz com que este não fique mais disponível como ponto para desfazer (`rollback`), mas não ocasiona nenhum outro comportamento visível pelo usuário. Não desfaz os efeitos dos comandos executados após o estabelecimento do ponto de salvamento (Para se fazer isto consulte *ROLLBACK TO SAVEPOINT*). A destruição de um ponto de salvamento quando não for mais necessário pode permitir ao sistema recuperar alguns recursos antes do fim da transação.

O comando `RELEASE SAVEPOINT` também destrói todos os pontos de salvamento estabelecidos depois que o ponto de salvamento especificado foi estabelecido.

## Parâmetros

*ponto\_de\_salvamento*

O nome do ponto de salvamento a ser destruído.

## Observações

Especificar um nome de ponto de salvamento que não foi definido anteriormente é um erro.

Não é possível liberar um ponto de salvamento quando a transação está em um estado interrompido (`aborted`).

Se vários pontos de salvamento tiverem o mesmo nome, somente o que foi definido mais recentemente será liberado.

## Exemplos

Para estabelecer um ponto de salvamento e destruí-lo posteriormente:

```
BEGIN;  
    INSERT INTO tabela1 VALUES (3);  
    SAVEPOINT meu_ponto_de_salvamento;  
    INSERT INTO tabela1 VALUES (4);  
    RELEASE SAVEPOINT meu_ponto_de_salvamento;  
COMMIT;
```

A transação acima insere tanto o 3 quanto o 4.

## Compatibilidade

Este comando está em conformidade com o padrão SQL:2003. O padrão especifica que a palavra chave `SAVEPOINT` é obrigatória, mas o PostgreSQL permite que seja omitida.

## Consulte também

*BEGIN, COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT, SAVEPOINT*

# RESET

## Nome

RESET — redefine o valor de um parâmetro em tempo de execução com seu valor padrão

## Sinopse

```
RESET nome
RESET ALL
```

## Descrição

O comando `RESET` redefine parâmetros em tempo de execução, colocando em cada um deles o seu respectivo valor padrão. O comando `RESET` é um forma alternativa de escrever

```
SET parâmetro TO DEFAULT
```

Consulte o comando `SET` para obter detalhes.

O valor padrão é definido como sendo o valor que o parâmetro deveria ter, se nenhum comando `SET` tivesse sido executado para este parâmetro na sessão corrente. A origem do valor padrão pode ser o valor padrão de compilação, o arquivo de configuração, opções da linha de comando, ou definições por banco de dados ou por usuário. Consulte a Seção 16.4 para obter detalhes.

Consulte a página de referência do comando `SET` para obter detalhes sobre o comportamento em transações do comando `RESET`.

## Parâmetros

*nome*

O nome de um parâmetro em tempo de execução. Consulte o comando `SET` para ver a relação.

ALL

Redefine todos os parâmetros em tempo de execução, que podem ser definidos, colocando em cada um deles o seu respectivo valor padrão.

## Exemplos

Definir a variável de configuração `geqo` com seu valor padrão:

```
RESET geqo;
```

## Compatibilidade

O comando `RESET` é uma extensão do PostgreSQL.



# REVOKE

## Nome

REVOKE — revoga privilégios de acesso

## Sinopse

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] nome_da_tabela [, ...]
    FROM { nome_do_usuario | GROUP nome_do_grupo | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { { CREATE | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE nome_do_banco_de_dados [, ...]
    FROM { nome_do_usuario | GROUP nome_do_grupo | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTION nome_da_função ([tipo, ...]) [, ...]
    FROM { nome_do_usuario | GROUP nome_do_grupo | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE nome_da_linguagem [, ...]
    FROM { nome_do_usuario | GROUP nome_do_grupo | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA nome_do_esquema [, ...]
    FROM { nome_do_usuario | GROUP nome_do_grupo | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE nome_do_espaco_de_tabelas [, ...]
    FROM { nome_do_usuario | GROUP nome_do_grupo | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

## Descrição

O comando REVOKE revoga, de um ou mais usuários ou grupos de usuários, privilégios concedidos anteriormente. A palavra chave PUBLIC se refere ao grupo contendo todos os usuários, definido implicitamente.

Veja na descrição do comando *GRANT* o significado de cada de privilégio.

Deve ser observado que um determinado usuário possui a soma dos privilégios concedidos diretamente para o próprio usuário mais os privilégios concedidos para os grupos dos quais é membro no momento, mais os privilégios concedidos para PUBLIC. Daí, por exemplo, revogar o privilégio SELECT de PUBLIC não significa, necessariamente, que todos os usuários perdem o privilégio SELECT para o objeto: aqueles que receberam o privilégio diretamente, ou por meio de um grupo, permanecem com o privilégio.

Se for especificado GRANT OPTION FOR somente a opção de concessão do privilégio é revogada, e não o próprio privilégio. Se não for especificado, tanto o privilégio quanto a opção de concessão são revogados.

Se o usuário possui um privilégio com opção de concessão, e concedeu este privilégio para outros usuários, então os privilégios que estes outros usuários possuem são chamados de privilégios dependentes. Se o privilégio ou a opção de concessão que o primeiro usuário possui for revogada, e existirem privilégios dependentes, estes privilégios dependentes também são revogados se for especificado *CASCADE*, senão a ação de revogar falha. Esta revogação recursiva somente afeta os privilégios que foram concedidos através de uma cadeia de usuários começando pelo usuário objeto deste comando *REVOKE*. Portanto, os usuários afetados poderão manter o privilégio se este privilégio também foi concedido por outros usuários.

## Observações

Use o comando `\z` do aplicativo *psql* para ver os privilégios concedidos para os objetos existentes. Consulte, também, o comando *GRANT* para obter informações sobre o formato.

Um determinado usuário pode revogar somente os privilégios que foram concedidos diretamente por este usuário. Se, por exemplo, o usuário A conceder um privilégio com opção de concessão para o usuário B, e o usuário B por sua vez conceder o privilégio para o usuário C, então o usuário A não poderá revogar diretamente o privilégio de C. Em vez disso, o usuário A poderá revogar a opção de concessão do usuário B e usar a opção *CASCADE*, para que o privilégio seja por sua vez revogado do usuário C. Outro exemplo é o caso em que tanto A quanto B concedem o mesmo privilégio a C, neste caso A pode revogar sua própria concessão mas não a concessão de B e, portanto, C continua com a concessão mesmo que A a revogue.

Quando alguém, que não é dono do objeto, tenta revogar privilégios para o objeto, o comando falha imediatamente se o usuário não possuir algum privilégio para o objeto. Desde que algum privilégio esteja disponível o comando prossegue, mas só revoga os privilégios para os quais o usuário tem a opção de concessão. A forma *REVOKE ALL PRIVILEGES* emite uma mensagem de advertência se nenhuma opção de concessão for possuída, enquanto as outras formas emitem mensagem de advertência se a opção de concessão para algum dos privilégios especificamente nomeados no comando não for possuída (Em princípio estas declarações também se aplicam ao dono do objeto, mas como o dono é sempre tratado como possuindo todas as opções de concessão estes casos nunca ocorrem).

Se um superusuário decidir executar o comando *GRANT* ou o comando *REVOKE*, o comando é executado como se tivesse sido executado pelo dono do objeto afetado. Uma vez que todos os privilégios partem do dono do objeto (possivelmente de forma indireta através de cadeias de opções de concessão), um superusuário pode revogar todos os privilégios, mas pode ser necessário usar *CASCADE* conforme mostrado acima.

## Exemplos

Revogar o privilégio de inserção na tabela *filmes* concedido para todos os usuários:

```
REVOKE INSERT ON filmes FROM PUBLIC;
```

Revogar todos os privilégios concedidos ao usuário *manuel* para a visão *vis\_tipos*:

```
REVOKE ALL PRIVILEGES ON vis_tipos FROM manuel;
```

Deve ser observado que este comando significa, na verdade, “revogue todos os privilégios que eu concedi”.

## Compatibilidade

As notas sobre compatibilidade presentes no comando *GRANT* se aplicam de forma análoga ao comando *REVOKE*. O sumário da sintaxe é:

```
REVOKE [ GRANT OPTION FOR ] privilégios
      ON objeto [ ( coluna [, ...] ) ]
      FROM { PUBLIC | nome_do_usuario [, ...] }
      { RESTRICT | CASCADE }
```

Um entre *RESTRICT* ou *CASCADE* é necessário de acordo com o padrão, mas o PostgreSQL assume *RESTRICT* por padrão.

## Consulte também

*GRANT*

# ROLLBACK

## Nome

ROLLBACK — interrompe a transação corrente

## Sinopse

```
ROLLBACK [ WORK | TRANSACTION ]
```

## Descrição

O comando `ROLLBACK` desfaz a transação corrente, fazendo com que todas as modificações realizadas por esta transação sejam desconsideradas.

## Parâmetros

`WORK`

`TRANSACTION`

Palavras chave opcionais. Não produzem nenhum efeito.

## Observações

Use o comando *COMMIT* para terminar uma transação bem-sucedida.

A utilização do `ROLLBACK` fora de uma transação não causa nenhum problema, mas causa uma mensagem de advertência.

## Exemplos

Para anular todas as modificações:

```
ROLLBACK ;
```

## Compatibilidade

O padrão SQL somente especifica as duas formas `ROLLBACK` e `ROLLBACK WORK`. Fora isso, este comando está em conformidade total.

## Consulte também

*BEGIN, COMMIT, ROLLBACK TO SAVEPOINT*

# ROLLBACK TO SAVEPOINT

## Nome

ROLLBACK TO SAVEPOINT — desfaz até o ponto de salvamento

## Sinopse

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] ponto_de_salvamento
```

## Descrição

Desfaz todos os comandos que foram executados após o estabelecimento do ponto de salvamento. O ponto de salvamento permanece válido, sendo possível desfazer até este ponto de salvamento novamente se for necessário.

O comando ROLLBACK TO SAVEPOINT destrói, implicitamente, todos os pontos de salvamento que foram estabelecidos após o ponto de salvamento especificado.

## Parâmetros

*ponto\_de\_salvamento*

O ponto de salvamento até onde será desfeito.

## Observações

Deve ser utilizado o comando *RELEASE SAVEPOINT* para destruir um ponto de salvamento sem descartar os efeitos dos comandos executados após este ponto de salvamento ter sido estabelecido.

A especificação de um nome de ponto de salvamento que não tenha sido estabelecido anteriormente é um erro.

Os cursores possuem um comportamento um tanto não transacional com respeito aos pontos de salvamento. Os cursores abertos dentro de um ponto de salvamento não são fechados quando se desfaz até o ponto de salvamento. Se um cursor for afetado pelo comando *FETCH* dentro de um ponto de salvamento, e a transação for desfeita até o ponto de salvamento, a posição do cursor permanece onde o comando *FETCH* a deixou; ou seja, o comando *FETCH* não é desfeito. Um cursor cuja execução tenha causado a interrupção da transação (*abort*) é colocado no estado de não poder executar e, portanto, embora a transação possa ser restaurada utilizando o comando ROLLBACK TO SAVEPOINT o cursor não pode mais ser usado.

## Exemplos

Para desfazer os efeitos dos comandos executados após o estabelecimento do *meu\_ponto\_de\_salvamento*:

```
ROLLBACK TO SAVEPOINT meu_ponto_de_salvamento;
```

As posições dos cursores não são afetadas por desfazer até o ponto de salvamento:

```
BEGIN;
```

```
DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;
```

```
SAVEPOINT foo;
```

```
FETCH 1 FROM foo;
```

```
  ?column?  
-----  
         1
```

```
ROLLBACK TO SAVEPOINT foo;
```

```
FETCH 1 FROM foo;
```

```
?column?
-----
2
```

```
COMMIT;
```

## Compatibilidade

O padrão SQL:2003 especifica que a palavra chave `SAVEPOINT` é obrigatória, mas o PostgreSQL e o Oracle <sup>1</sup> permitem que seja omitida. O padrão SQL:2003 permite apenas `WORK`, e não `TRANSACTION`, como palavra sem efeito (*noise word*) após o `ROLLBACK`. Além disso, o padrão SQL:2003 possui a cláusula opcional `AND [ NO ] CHAIN` que não é suportada pelo PostgreSQL atualmente. Fora isto, este comando está em conformidade com o padrão SQL.

## Consulte também

*BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, SAVEPOINT*

## Notas

1. No Oracle a utilização do comando `ROLLBACK` com a cláusula `TO SAVEPOINT` realiza as seguintes operações: desfaz apenas a parte da transação após o ponto de salvamento; apaga todos os pontos de salvamento criados após este ponto de salvamento; libera todos os bloqueios em tabelas e linhas obtidos desde o ponto de salvamento; as outras transações que requisitaram acesso às linhas bloqueadas após este ponto de salvamento devem continuar aguardando até que a transação seja efetivada ou desfeita; as outras transações que ainda não requisitaram as linhas podem requisitar e acessar as linhas imediatamente. `ROLLBACK`  
([http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/statements\\_919a.htm#2104639](http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/statements_919a.htm#2104639)) (N. do T.)

# SAVEPOINT

## Nome

`SAVEPOINT` — define um novo ponto de salvamento dentro da transação corrente

## Sinopse

```
SAVEPOINT ponto_de_salvamento
```

## Descrição

O comando `SAVEPOINT` estabelece um novo ponto de salvamento dentro da transação corrente.

O ponto de salvamento é uma marca especial dentro da transação que permite desfazer todos os comandos executados após o seu estabelecimento, restaurando o estado da transação ao que era quando o ponto de salvamento foi estabelecido <sup>1</sup>.

## Parâmetros

*ponto\_de\_salvamento*

O nome a ser dado ao novo ponto de salvamento.

## Observações

Deve ser utilizado o comando `ROLLBACK TO SAVEPOINT` para desfazer até o ponto de salvamento. Deve ser utilizado o comando `RELEASE SAVEPOINT` para destruir um ponto de salvamento, porém mantendo os efeitos dos comandos executados após este ter sido estabelecido.

Os pontos de salvamento somente podem ser estabelecidos dentro de um bloco de transação. Podem haver vários pontos de salvamento definidos dentro de uma transação.

## Exemplos

Para estabelecer um ponto de salvamento e, posteriormente, desfazer o efeito de todos os comandos executados após o seu estabelecimento:

```
BEGIN;  
    INSERT INTO tabela1 VALUES (1);  
    SAVEPOINT meu_ponto_de_salvamento;  
    INSERT INTO tabela1 VALUES (2);  
    ROLLBACK TO SAVEPOINT meu_ponto_de_salvamento;  
    INSERT INTO tabela1 VALUES (3);  
COMMIT;
```

A transação acima insere os valores 1 e 3, mas não o 2.

Para estabelecer e, posteriormente, destruir um ponto de salvamento:

```
BEGIN;  
    INSERT INTO tabela1 VALUES (3);  
    SAVEPOINT meu_ponto_de_salvamento;  
    INSERT INTO tabela1 VALUES (4);  
    RELEASE SAVEPOINT meu_ponto_de_salvamento;  
COMMIT;
```

A transação acima insere tanto o 3 quanto o 4.

## Compatibilidade

O padrão SQL requer que um ponto de salvamento seja destruído, automaticamente, quando é estabelecido um outro ponto de salvamento com o mesmo nome. No PostgreSQL o ponto de salvamento é mantido, embora somente o mais recente seja

utilizado ao se desfazer ou liberar; a liberação do ponto de salvamento mais novo torna o ponto de salvamento mais antigo acessível novamente para os comandos `ROLLBACK TO SAVEPOINT` e `RELEASE SAVEPOINT`. Fora isso, o comando `SAVEPOINT` está em conformidade total com o padrão SQL.

## Consulte também

*BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, ROLLBACK TO SAVEPOINT*

## Notas

1. Pontos de Salvamento — os pontos de salvamento dividem uma transação longa, com muitos comandos SQL, em partes menores. Com os pontos de salvamento, o trabalho pode ser marcado arbitrariamente em qualquer ponto dentro de uma transação longa. Isto dá a opção de, mais tarde, desfazer todo o trabalho realizado do ponto corrente na transação até o ponto de salvamento declarado dentro da transação. Por exemplo, podem ser utilizados pontos de salvamento através de uma série longa e complexa de atualizações e, então, se acontecer algum erro não será necessário executar novamente todos os comandos. Introduction to the Oracle Server  
([http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96524/c01\\_02intro.htm#46926](http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96524/c01_02intro.htm#46926)) (N. do T.)

# SELECT

## Nome

SELECT — retorna linhas de uma tabela ou de uma visão

## Sinopse

```
SELECT [ ALL | DISTINCT [ ON ( expressão [, ...] ) ] ]
      * | expressão [ AS nome_de_saída ] [, ...]
[ FROM item_do_from [, ...] ]
[ WHERE condição ]
[ GROUP BY expressão [, ...] ]
[ HAVING condição [, ...] ]
[ { UNION | INTERSECT | EXCEPT } [ ALL ] seleção ]
[ ORDER BY expressão [ ASC | DESC | USING operador ] [, ...] ]
[ LIMIT { contador | ALL } ]
[ OFFSET início ]
[ FOR UPDATE [ OF nome_da_tabela [, ...] ] ]
```

onde *item\_do\_from* pode ser um entre:

```
[ ONLY ] nome_da_tabela [ * ] [ [ AS ] aliás [ ( aliás_de_coluna [, ...] ) ] ]
( seleção ) [ AS ] aliás [ ( aliás_de_coluna [, ...] ) ]
nome_da_função ( [ argumento [, ...] ] ) [ AS ] aliás [ ( aliás_de_coluna [, ...] |
definição_de_coluna [, ...] ) ]
nome_da_função ( [ argumento [, ...] ] ) AS ( definição_de_coluna [, ...] )
item_do_from [ NATURAL ] tipo_de_junção item_do_from [ ON condição_de_junção | USING (
coluna_de_junção [, ...] ) ]
```

## Descrição

O comando `SELECT` retorna linhas de uma ou mais tabelas. O processamento geral do comando `SELECT` está descrito abaixo:

1. Todos os elementos da lista `FROM` são computados; cada elemento na lista `FROM` é uma tabela real ou virtual. Quando é especificado mais de um elemento na lista `FROM`, é feita uma junção cruzada entre estes elementos (Consulte a *Cláusula FROM* abaixo).
2. Se for especificada a cláusula `WHERE`, todas as linhas que não satisfazem a condição são eliminadas da saída (Consulte a *Cláusula WHERE* abaixo).
3. Se for especificada a cláusula `GROUP BY`, a saída é dividida em grupos de linhas que correspondem a um ou mais valores. Se a cláusula `HAVING` estiver presente, são eliminados os grupos que não satisfazem a condição especificada (Consulte a *Cláusula GROUP BY* e a *Cláusula HAVING* abaixo).
4. As linhas de saída reais são computadas utilizando as expressões de saída do comando `SELECT` para cada linha selecionada (Consulte a *Lista do SELECT* abaixo).
5. Usando os operadores `UNION`, `INTERSECT` e `EXCEPT` podem ser combinadas as saídas de vários comandos `SELECT` para formar um único conjunto de resultados. O operador `UNION` retorna todas as linhas presentes em um ou nos dois conjuntos de resultados. O operador `INTERSECT` retorna todas as linhas presentes nos dois conjuntos de resultados. O operador `EXCEPT` retorna as linhas presentes no primeiro conjunto de resultados mas não no segundo. Em todos estes três casos as linhas duplicadas são eliminadas, a menos que `ALL` seja especificado (Consulte a *Cláusula UNION*, a *Cláusula INTERSECT* e *Cláusula EXCEPT* abaixo).
6. Se for especificada a cláusula `ORDER BY`, as linhas retornadas são classificadas segundo a ordem especificada. Se a cláusula `ORDER BY` não for especificada, as linhas são retornadas na ordem em que o sistema considerar mais rápido de produzir (Consulte a *Cláusula ORDER BY* abaixo).



7. A cláusula `DISTINCT` remove do resultado as linhas duplicadas <sup>1 2</sup> . A cláusula `DISTINCT ON` remove as linhas que correspondem a todas as expressões especificadas. A cláusula `ALL` (o padrão) retorna todas as linhas candidatas, incluindo as duplicadas (Consulte a *Cláusula `DISTINCT`* abaixo).
8. Se for especificada a cláusula `LIMIT` ou a cláusula `OFFSET`, o comando `SELECT` retorna somente um subconjunto das linhas do resultado (Consulte a *Cláusula `LIMIT`* abaixo) <sup>3</sup> .
9. A cláusula `FOR UPDATE` faz o comando `SELECT` bloquear as linhas selecionadas contra atualizações concorrentes (Consulte a *Cláusula `FOR UPDATE`* abaixo) <sup>4</sup> .

É necessário possuir o privilégio `SELECT` na tabela para poder ler seus valores. A utilização de `FOR UPDATE` requer também o privilégio `UPDATE`.

## Parâmetros

### Cláusula `FROM`

A cláusula `FROM` especifica uma ou mais tabelas fonte para o comando `SELECT`. Se várias fontes forem especificadas, o resultado será o produto cartesiano (junção cruzada) de todas as fontes, mas normalmente são incluídas condições de qualificação para restringir as linhas retornadas a um pequeno subconjunto do produto cartesiano.

A cláusula `FROM` pode conter os seguintes elementos:

#### *nome\_da\_tabela*

O nome (opcionalmente qualificado pelo esquema) de uma tabela ou de uma visão existente. Se a cláusula `ONLY` for especificada, somente esta tabela será varrida. Se a cláusula `ONLY` não for especificada, esta tabela e todas as suas tabelas descendentes (se existirem) serão varridas. Pode ser anexado um `*` ao nome da tabela para indicar que as tabelas descendentes devem ser varridas, mas na versão corrente este é o comportamento padrão; nas versões anteriores a 7.1 o comportamento padrão era `ONLY`. O comportamento padrão pode ser modificado mudando o valor da opção de configuração `sql_inheritance`.

#### *aliás*

Um nome substituto para o item da cláusula `FROM` contendo o aliás. O aliás é utilizado para abreviar, ou para eliminar ambigüidade em auto-junções (onde a mesma tabela é varrida várias vezes). Quando se fornece um aliás, o nome verdadeiro da tabela ou da função fica totalmente escondido; se, por exemplo, for declarado `FROM foo AS f`, o restante do comando `SELECT` deve fazer referência a este item do `FROM` como `f`, e não como `foo`. Se for escrito um aliás, também pode ser escrita uma lista de aliases de coluna para fornecer nomes substitutos para uma ou mais colunas da tabela.

#### *seleção*

Pode haver sub-`SELECT` na cláusula `FROM`. Atua como se sua saída fosse criada como uma tabela temporária pela duração deste único comando `SELECT`. Deve ser observado que o sub-`SELECT` deve estar entre parênteses, e que *deve* ser especificado um aliás para o mesmo.

#### *nome\_da\_função*

Podem estar presentes na cláusula `FROM` chamadas de função (É especialmente útil no caso das funções que retornam um conjunto de resultados, mas pode ser usada qualquer função). Atua como se a sua saída fosse criada como uma tabela temporária pela duração deste único comando `SELECT`. Também pode ser utilizado um aliás. Se for escrito um aliás, também pode ser escrita uma lista de aliases de coluna para fornecer nomes substitutos para um ou mais atributos do tipo composto retornado pela função. Se a função tiver sido definida como retornando o tipo de dado `record` então deve estar presente um aliás, ou a palavra chave `AS` seguida por uma lista de definições de coluna na forma ( `nome_de_coluna tipo_de_dado [ , ... ]` ). A lista de definições de coluna deve corresponder ao número e tipo reais das colunas retornadas pela função.

#### *tipo\_de\_junção*

Um entre

- `[ INNER ] JOIN`
- `LEFT [ OUTER ] JOIN`

- RIGHT [ OUTER ] JOIN
- FULL [ OUTER ] JOIN
- CROSS JOIN

Para os tipos de junção INNER e OUTER deve ser especificada uma condição de junção designando exatamente um entre NATURAL, ON *condição\_de\_junção* ou USING (*coluna\_de\_junção* [, ...]). Veja abaixo o significado. Para CROSS JOIN, nenhuma destas cláusulas pode estar presente.

A cláusula JOIN combina dois itens da cláusula FROM. Se for necessário devem ser utilizados parênteses para determinar a ordem de aninhamento. Na ausência de parênteses, a cláusula JOIN aninha da esquerda para a direita. Em todos os casos a cláusula JOIN tem nível de precedência superior ao das vírgulas que separam os itens da cláusula FROM.

CROSS JOIN e INNER JOIN produzem um produto cartesiano simples, o mesmo resultado obtido listando os dois itens no nível superior da cláusula FROM, mas restrito pela condição de junção (se houver). CROSS JOIN é equivalente a INNER JOIN ON (TRUE), ou seja, nenhuma linha é removida pela qualificação. Estes tipos de junção são apenas uma notação conveniente, uma vez que não fazem nada que não poderia ser feito usando simplesmente FROM e WHERE.

LEFT OUTER JOIN retorna todas as linhas presentes no produto cartesiano qualificado (ou seja, todas as linhas combinadas que passam pela sua condição de junção), mais uma cópia de cada linha da tabela à esquerda para a qual não há linha na tabela à direita que passe pela condição de junção. As linhas da tabela à esquerda são estendidas por toda a largura da tabela de junção, inserindo valores nulos para as colunas da tabela à direita. Deve ser observado que somente a condição da própria cláusula JOIN é considerada ao decidir quais linhas possuem correspondência. As condições externas são aplicadas depois.

De forma inversa, RIGHT OUTER JOIN retorna todas as linhas da junção, mais uma linha para cada linha da tabela à direita sem correspondência (estendida com nulos à esquerda). É apenas uma notação conveniente, uma vez que pode ser convertido em LEFT OUTER JOIN trocando as entradas à direita e à esquerda.

FULL OUTER JOIN retorna todas as linhas da junção, mais uma linha para cada linha da tabela à esquerda sem correspondência, (estendida com nulos à direita), mais uma linha para cada linha da tabela à direita sem correspondência (estendida com nulos à esquerda).

ON *condição\_de\_junção*

A *condição\_de\_junção* é uma expressão que resulta em um valor do tipo boolean (semelhante à cláusula WHERE) que especifica quais linhas da junção são consideradas correspondentes.

USING (*coluna\_de\_junção* [, ...])

A cláusula com a forma USING ( *a*, *b*, ... ) é uma abreviação de ON *tabela\_à\_esquerda*.*a* = *tabela\_à\_direita*.*a* AND *tabela\_à\_esquerda*.*b* = *tabela\_à\_direita*.*b* .... USING também implica que somente será incluída na saída da junção uma de cada par de colunas equivalentes, e não as duas.

NATURAL

NATURAL é uma forma abreviada para a lista USING mencionando todas as colunas das duas tabelas que possuem o mesmo nome.

## Cláusula WHERE

A cláusula opcional WHERE possui a forma geral

WHERE *condição*

onde *condição* é uma expressão que produz um resultado do tipo boolean <sup>5</sup>. Todas as linhas que não satisfazem a esta condição são eliminadas da saída. A linha satisfaz a condição se retorna verdade quando os valores reais da linha são colocados no lugar das variáveis que os referenciam.

## Cláusula GROUP BY

A cláusula opcional GROUP BY possui a forma geral

GROUP BY *expressão* [, ...]

A cláusula `GROUP BY` condensa em uma única linha todas as linhas selecionadas que compartilham os mesmos valores para as expressões de agrupamento. A *expressão* pode ser o nome de uma coluna da entrada, ou o nome ou o número ordinal de uma coluna da saída (lista de itens do `SELECT`), ou uma expressão arbitrária formada por valores das colunas da entrada. Havendo ambigüidade, o nome na cláusula `GROUP BY` será interpretado como sendo o nome da coluna da entrada, e não o nome da coluna da saída.

As funções de agregação, caso sejam usadas, são computadas entre todas as linhas que constituem cada grupo, produzindo um valor separado para cada grupo (enquanto sem `GROUP BY`, uma agregação produz um único valor computado entre todas as linhas selecionadas). Quando `GROUP BY` está presente, não é válido a lista de expressões do `SELECT` fazer referência a colunas não agrupadas, exceto dentro das funções de agregação, uma vez que haveria mais de um valor possível retornado para uma coluna não agrupada.

### Cláusula **HAVING**

A cláusula opcional `HAVING` possui a forma geral

`HAVING condição`

onde *condição* é especificada da mesma forma que na cláusula `WHERE`.

A cláusula `HAVING` elimina os grupos de linhas que não satisfazem a condição. A cláusula `HAVING` é diferente da cláusula `WHERE`: `WHERE` filtra individualmente as linhas antes do `GROUP BY` ser aplicado, enquanto `HAVING` filtra grupos de linhas criados pelo `GROUP BY`. Cada coluna referenciada na *condição* deve referenciar sem ambigüidade uma coluna de agrupamento, a menos que a referência apareça dentro de uma função de agregação.

A presença da cláusula `HAVING` torna a consulta uma consulta agrupada, mesmo que não exista a cláusula `GROUP BY`. É o mesmo que acontece quando a consulta contém funções de agregação mas não possui a cláusula `GROUP BY`. Todas as linhas selecionadas são consideradas como formando um único grupo, e tanto a lista do `SELECT` quanto a cláusula `HAVING` somente podem fazer referência a colunas da tabela dentro de funções de agregação. Este tipo de consulta gera uma única linha se a condição da cláusula `HAVING` for verdade, ou nenhuma linha se não for verdade.

### Lista do **SELECT**

A lista do `SELECT` (entre as palavras chave `SELECT` e `FROM`) especifica expressões que formam as linhas de saída do comando `SELECT`. As expressões podem (e geralmente fazem) referenciar colunas computadas na cláusula `FROM`. Usando a cláusula `AS nome_de_saída`, pode ser especificado outro nome para uma coluna da saída. Este nome é usado, principalmente, como rótulo da coluna mostrada. Também pode ser usado para fazer referência ao valor da coluna nas cláusulas `ORDER BY` e `GROUP BY`, mas não nas cláusulas `WHERE` e `HAVING`; nestas, a expressão deve ser escrita.

Em vez da expressão pode ser escrito `*` na lista de saída, como abreviação para todas as colunas das linhas selecionadas. Também pode ser escrito `nome_da_tabela.*` como abreviação das colunas provenientes apenas desta tabela.

### Cláusula **UNION**

A cláusula `UNION` possui a forma geral

`comando_de_seleção UNION [ ALL ] comando_de_seleção`

onde *comando\_de\_seleção* é qualquer comando `SELECT` sem as cláusulas `ORDER BY`, `LIMIT` e `FOR UPDATE` (as cláusulas `ORDER BY` e `LIMIT` podem ser aplicadas a uma subexpressão se esta estiver entre parênteses. Sem os parênteses, estas cláusulas são consideradas como aplicadas ao resultado da cláusula `UNION`, e não à sua expressão de entrada à direita).

O operador `UNION` computa o conjunto formado pela união das linhas retornadas pelos comandos `SELECT` envolvidos. Uma linha está presente no conjunto união dos dois conjuntos de resultados se estiver presente em pelo menos um destes dois conjuntos de resultados. Os dois comandos `SELECT` que representam os operandos diretos do operador `UNION` devem produzir o mesmo número de colunas, e as colunas correspondentes devem possuir tipos de dado compatíveis.

O resultado do operador `UNION` não contém nenhuma linha duplicada, a menos que a opção `ALL` seja especificada. `ALL` não permite a eliminação das duplicatas; portanto, `UNION ALL` geralmente é significativamente mais rápido do que `UNION`; deve ser utilizado `ALL` se possível.

Havendo vários operadores UNION no mesmo comando SELECT, estes são avaliados da esquerda para a direita, a menos que os parênteses indiquem o contrário.

Atualmente não pode ser especificado FOR UPDATE nem para o resultado do operador UNION nem para qualquer entrada do operador UNION.

### Cláusula INTERSECT

A cláusula INTERSECT possui a forma geral

*comando\_de\_seleção* INTERSECT [ ALL ] *comando\_de\_seleção*

onde *comando\_de\_seleção* é qualquer comando SELECT sem as cláusulas ORDER BY, LIMIT e FOR UPDATE.

O operador INTERSECT computa o conjunto formado pela interseção das linhas retornadas pelos comandos SELECT envolvidos. Uma linha está na interseção dos dois conjuntos de resultados se estiver presente nos dois conjuntos de resultados.

O resultado do operador INTERSECT não contém nenhuma linha duplicada, a menos que a opção ALL seja especificada. Usando ALL, uma linha contendo  $m$  duplicatas na tabela à esquerda e  $n$  duplicatas na tabela à direita, aparece  $\min(m,n)$  vezes no conjunto de resultados.

Havendo vários operadores INTERSECT no mesmo comando SELECT, estes são avaliados da esquerda para a direita, a menos que os parênteses indiquem outra ordem. O operador INTERSECT tem nível de precedência superior ao do operador UNION, ou seja, A UNION B INTERSECT C é lido como A UNION (B INTERSECT C).

Atualmente a cláusula FOR UPDATE não pode ser especificada para o resultado do operador INTERSECT, nem em qualquer entrada do operador INTERSECT.

### Cláusula EXCEPT

A cláusula EXCEPT possui a forma geral

*comando\_de\_seleção* EXCEPT [ ALL ] *comando\_de\_seleção*

onde *comando\_de\_seleção* é qualquer comando SELECT sem as cláusulas ORDER BY, LIMIT e FOR UPDATE.

O operador EXCEPT computa o conjunto de linhas presentes no resultado do comando SELECT à esquerda, mas que não estão presentes no resultado do comando à direita.

O resultado do operador EXCEPT não contém nenhuma linha duplicada, a menos que a cláusula ALL seja especificada. Usando ALL, uma linha que possua  $m$  duplicatas na tabela à esquerda e  $n$  duplicatas na tabela à direita aparece  $\max(m-n,0)$  vezes no conjunto de resultados.

Havendo vários operadores EXCEPT no mesmo comando SELECT, estes são processados da esquerda para a direita, a menos que os parênteses especifiquem outra ordem. O operador EXCEPT possui o mesmo nível de precedência do operador UNION.

Atualmente a cláusula FOR UPDATE não pode ser especificada para o resultado do operador EXCEPT, nem em nenhuma entrada do operador EXCEPT.

### Cláusula ORDER BY

A cláusula opcional ORDER BY possui a forma geral

ORDER BY *expressão* [ ASC | DESC | USING *operador* ] [, ...]

onde *expressão* pode ser o nome ou o número ordinal de uma coluna da saída (item da lista do SELECT), ou pode ser uma expressão arbitrária formada por valores das colunas da entrada.

A cláusula ORDER BY faz as linhas do resultado serem classificadas de acordo com as expressões especificadas. Se duas linhas são iguais de acordo com a expressão mais à esquerda, estas são comparadas de acordo com a próxima expressão, e assim por diante. Se forem iguais de acordo com todas as expressões especificadas, são retornadas em uma ordem dependente da implementação.

O número ordinal se refere à posição ordinal (esquerda para a direita) da coluna do resultado. Esta funcionalidade torna possível definir uma ordenação baseada em uma coluna que não possui um nome único. Isto nunca é absolutamente necessário, porque sempre é possível atribuir um nome à coluna do resultado usando a cláusula `AS`.

Também é possível utilizar expressões arbitrárias na cláusula `ORDER BY`, incluindo colunas que não aparecem na lista de resultado do `SELECT`. Portanto, o seguinte comando é válido:

```
SELECT nome FROM distribuidores ORDER BY codigo;
```

A limitação desta funcionalidade é que a cláusula `ORDER BY` aplicada ao resultado de `UNION`, `INTERSECT` ou `EXCEPT` pode especificar apenas nomes de coluna ou números, mas não expressões.

Se a expressão no `ORDER BY` for simplesmente um nome correspondendo tanto ao nome de uma coluna do resultado quanto ao nome de uma coluna da entrada, o `ORDER BY` interpreta como sendo o nome da coluna do resultado. Esta é a escolha oposta à feita pelo `GROUP BY` na mesma situação. Esta incoerência existe para ficar compatível com o padrão SQL.

Pode ser adicionada, opcionalmente, a palavra chave `ASC` (ascendente) ou `DESC` (descendente) após cada expressão na cláusula `ORDER BY`. Se nenhuma das duas for especificada, `ASC` é assumido por padrão. Como alternativa, pode ser especificado o nome de um operador de ordenação específico na cláusula `USING`. Geralmente `ASC` é equivalente a `USING <` e geralmente `DESC` é equivalente a `USING >` (Mas o criador de um tipo de dado definido pelo usuário pode definir exatamente qual é a ordem de classificação padrão, podendo corresponder a operadores com outros nomes).

O valor nulo é classificado em uma posição mais alta do que qualquer outro valor. Em outras palavras, na ordem de classificação ascendente os valores nulos ficam no final, e na ordem de classificação descendente os valores nulos ficam no início.<sup>6</sup>

Dados na forma de cadeias de caracteres são classificados de acordo com a ordem de classificação (`collation`<sup>7</sup>) estabelecida quando o agrupamento de bancos de dados foi inicializado.

## Cláusula `DISTINCT`

Se for especificada a cláusula `DISTINCT`, todas as linhas duplicadas são removidas do conjunto de resultados (é mantida uma linha para cada grupo de duplicatas). A cláusula `ALL` especifica o oposto: todas as linhas são mantidas; este é o padrão.

`DISTINCT ON ( expressão [, ...] )` preserva apenas a primeira linha de cada conjunto de linhas onde as expressões fornecidas forem iguais. As expressões em `DISTINCT ON` são interpretadas usando as mesmas regras da cláusula `ORDER BY` (veja acima). Deve ser observado que a “primeira linha” de cada conjunto é imprevisível, a menos que seja utilizado `ORDER BY` para garantir que a linha desejada apareça na frente. Por exemplo,

```
SELECT DISTINCT ON (local) local, data, condicao
FROM tbl_condicao_climatica
ORDER BY local, data DESC;
```

mostra o relatório de condição climática mais recente para cada local, mas se não tivesse sido usado `ORDER BY` para obrigar a ordem descendente dos valores da data para cada local, teria sido obtido um relatório com datas imprevisíveis para cada local.

As expressões em `DISTINCT ON` devem corresponder às expressões mais à esquerda no `ORDER BY`. A cláusula `ORDER BY` normalmente contém expressões adicionais para determinar a precedência desejada das linhas dentro de cada grupo `DISTINCT ON`.

## Cláusula `LIMIT`

A cláusula `LIMIT` consiste em duas subcláusulas independentes:

```
LIMIT { contador | ALL }
OFFSET início
```

onde *contador* especifica o número máximo de linhas a serem retornadas, enquanto *início* especifica o número de linhas a serem puladas antes de começar a retornar as linhas. Quando as duas são especificadas, as *início* primeiras linhas são puladas antes de começar a contar as *contador* linhas a serem retornadas.

Ao se usar a cláusula `LIMIT` é uma boa idéia usar também a cláusula `ORDER BY` para colocar as linhas do resultado dentro de uma ordem única. Caso contrário será retornado um subconjunto imprevisível de linhas da consulta — pode-se estar

querendo receber da décima a vigésima linha, mas da décima a vigésima em que ordem? Não é possível saber qual será a ordem, a não ser que `ORDER BY` seja especificado.

O planejador de comandos leva `LIMIT` em consideração ao gerar o plano da consulta, por isso é muito provável serem obtidos planos diferentes (produzindo linhas em ordens diferentes) dependendo do que for especificado para `LIMIT` e `OFFSET`. Portanto, utilizar valores diferentes para `LIMIT/OFFSET` para selecionar subconjuntos diferentes do resultado da consulta *produz resultados inconsistentes*, a não ser que seja exigida uma ordem previsível para os resultados utilizando `ORDER BY`. Isto não está errado; isto é uma consequência direta do fato do SQL não prometer retornar os resultados de uma consulta em nenhuma ordem específica, a não ser que `ORDER BY` seja utilizado para impor esta ordem.

## Cláusula FOR UPDATE

A cláusula `FOR UPDATE` possui a forma

```
FOR UPDATE [ OF nome_da_tabela [, ...] ]
```

A cláusula `FOR UPDATE` faz as linhas selecionadas pelo comando `SELECT` serem bloqueadas como se fosse para atualização. Isto impede a modificação ou exclusão destas linhas por outras transações até a transação corrente terminar. Ou seja, outras transações tentando usar os comandos `UPDATE`, `DELETE` ou `SELECT FOR UPDATE` nestas linhas ficam bloqueadas até a transação corrente terminar. Também, se um comando `UPDATE`, `DELETE`, ou `SELECT FOR UPDATE` de outra transação já tiver bloqueado uma ou várias destas linhas, o `SELECT FOR UPDATE` fica aguardando a outra transação completar e, então, bloqueia e retorna a linha atualizada (ou nenhuma linha, se a linha foi excluída). Para obter mais explicações consulte o Capítulo 12.

Se forem especificados nomes de tabelas na cláusula `FOR UPDATE`, então somente as linhas oriundas destas tabelas são bloqueadas; todas as outras tabelas usadas no `SELECT` são simplesmente lidas como de costume.

`FOR UPDATE` não pode ser utilizado nos contextos onde as linhas retornadas não podem ser claramente identificadas com as linhas individuais da tabela; por exemplo, não pode ser utilizado junto com agregações.

`FOR UPDATE` pode estar antes de `LIMIT` para manter a compatibilidade com as versões do PostgreSQL anteriores a 7.3. Entretanto, será executado após o `LIMIT` e, portanto, este é o lugar adequado para ser escrito.

## Exemplos

Para efetuar a junção da tabela `filmes` com a tabela `distribuidores`:

```
SELECT f.titulo, f.did, d.nome, f.data_prod, f.tipo
FROM distribuidores d, filmes f
WHERE f.did = d.did
```

titulo	did	nome	data_prod	tipo
The Third Man	101	British Lion	1949-12-23	Drama
The African Queen	101	British Lion	1951-08-11	Romantic
...				

Para somar a coluna `duracao` de todos os filmes, e agrupar os resultados por `tipo`:

```
SELECT tipo, sum(duracao) AS total FROM filmes GROUP BY tipo;
```

tipo	total
Action	07:34
Comedy	02:58
Drama	14:28
Musical	06:42
Romantic	04:38

Para somar a coluna `duracao` de todos os filmes, agrupar os resultados por `tipo`, e mostrar apenas os grupos com total inferior a 5 horas:

```
SELECT tipo, sum(duracao) AS total
FROM filmes
GROUP BY tipo
HAVING sum(duracao) < interval '5 hours';
```

tipo	total
Comédia	02:58
Romance	04:38

Os dois exemplos a seguir são formas idênticas de classificação dos resultados individuais de acordo com o conteúdo da segunda coluna (nome):

```
SELECT * FROM distribuidores ORDER BY nome;
SELECT * FROM distribuidores ORDER BY 2;
```

did	nome
109	20th Century Fox
110	Bavaria Atelier
101	British Lion
107	Columbia
102	Jean Luc Godard
113	Luso films
104	Mosfilm
103	Paramount
106	Toho
105	United Artists
111	Walt Disney
112	Warner Bros.
108	Westward

O próximo exemplo mostra como obter a união da tabela distribuidores com a tabela atores, restringindo o resultado aos nomes que iniciam pela letra W em cada uma das tabelas. Somente são desejadas linhas distintas, por isso a palavra chave ALL é omitida:

distribuidores		atores:	
did	nome	id	nome
108	Westward	1	Woody Allen
111	Walt Disney	2	Warren Beatty
112	Warner Bros.	3	Walter Matthau
...		...	

```
SELECT distribuidores.nome
FROM distribuidores
WHERE distribuidores.nome LIKE 'W%'
UNION
SELECT atores.nome
FROM atores
WHERE atores.nome LIKE 'W%';
```

nome
Walt Disney
Walter Matthau
Warner Bros.
Warren Beatty
Westward
Woody Allen

Este exemplo mostra como usar uma função na cláusula FROM, com e sem uma lista de definição de colunas:

```
CREATE FUNCTION distribuidores(int) RETURNS SETOF distribuidores AS $$
    SELECT * FROM distribuidores WHERE did = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM distribuidores(111);
```

```

did |      nome
-----+-----
111 | Walt Disney
```

```
CREATE FUNCTION distribuidores_2(int) RETURNS SETOF record AS $$
    SELECT * FROM distribuidores WHERE did = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM distribuidores_2(111) AS (f1 int, f2 text);
```

```

f1 |      f2
-----+-----
111 | Walt Disney
```

## Compatibilidade

Obviamente, o comando SELECT é compatível com o padrão SQL. Entretanto, existem algumas extensões e algumas funcionalidades faltando.

### Cláusula FROM omitida

O PostgreSQL permite omitir a cláusula FROM. Tem uso direto no cômputo de resultados de expressões simples:

```
SELECT 2+2;
```

```

?column?
-----
4
```

Alguns outros bancos de dados SQL não podem fazer isto, a não ser introduzindo uma tabela fictícia de uma linha para executar o comando SELECT.

Uma utilização menos óbvia desta funcionalidade é abreviar comandos SELECT comuns de tabelas:

```
SELECT distribuidores.* WHERE distribuidores.nome = 'Westward';
```

```

did |      nome
-----+-----
108 | Westward
```

Isso funciona porque é adicionado um item implícito no FROM para cada tabela que é referenciada nas outras partes do comando SELECT, mas que não é mencionada no FROM.

Ao mesmo tempo em que é uma forma conveniente de abreviar, é fácil ser usado incorretamente. Por exemplo, o comando

```
SELECT distribuidores.* FROM distribuidores d;
```

provavelmente deve ser um engano; é mais provável que se deseje

```
SELECT d.* FROM distribuidores d;
```

do que a junção sem restrições

```
SELECT distribuidores.* FROM distribuidores d, distribuidores distribuidores;
```



que seria obtida na verdade. Para ajudar a detectar este tipo de engano, o PostgreSQL adverte se uma funcionalidade `FROM` implícita é utilizada em um comando `SELECT` que também contenha uma cláusula `FROM` explícita. Também é possível desabilitar a funcionalidade de `FROM`-implícito definindo a parâmetro `add_missing_from` como falso.

## A palavra chave `AS`

No padrão SQL, a palavra chave opcional `AS` é sem efeito, podendo ser omitida sem afetar o significado. O analisador do PostgreSQL requer esta palavra chave quando uma coluna da saída é renomeada, porque a funcionalidade de extensividade de tipo conduz o analisador a ambigüidades caso não esteja presente. Entretanto, `AS` é opcional nos itens do `FROM`.

## Espaço de nomes disponível para `GROUP BY` e `ORDER BY`

No padrão SQL-92, a cláusula `ORDER BY` somente pode utilizar nomes ou números das colunas do resultado, enquanto a cláusula `GROUP BY` somente pode utilizar expressões baseadas nos nomes das colunas da entrada. O PostgreSQL estende estas duas cláusulas para permitir, também, a outra escolha (mas utiliza a interpretação padrão se houver ambigüidade). O PostgreSQL também permite que as duas cláusulas especifiquem expressões arbitrárias. Deve ser observado que os nomes que aparecem na expressão são sempre considerados como nomes das colunas da entrada, e não como nomes das colunas do resultado.

O SQL:1999 utiliza uma definição um pouco diferente, que não é inteiramente compatível com o SQL-92. Entretanto, na maioria dos casos o PostgreSQL interpreta uma expressão presente em `ORDER BY` ou `GROUP BY` da mesma maneira que o SQL:1999.

## Cláusulas fora do padrão

As cláusulas `DISTINCT ON`, `LIMIT` e `OFFSET` não são definidas no padrão SQL.

## Notas

1. Oracle — deve ser especificado `DISTINCT` ou `UNIQUE` se for desejado que o Oracle retorne apenas uma cópia de cada conjunto de linhas duplicadas selecionadas (esta duas palavras chave são sinônimos). As linhas duplicadas são aquelas com valores correspondentes para cada expressão na lista de seleção; `DISTINCT` não pode ser especificado quando a lista de seleção contém colunas `LOB`. Oracle9i SQL Reference - Release 2 (9.2) - Part Number A96540-02 ([http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/statements\\_103a.htm#2065826](http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/statements_103a.htm#2065826)) (N. do T.)
2. SQL Server — `DISTINCT` especifica que somente linhas únicas podem aparecer no conjunto de resultados. Os valores nulos são considerados iguais para as finalidades da palavra chave `DISTINCT`. SQL Server Books Online (N. do T.)
3. SQL Server — `TOP n [PERCENT]` especifica que somente as primeiras `n` linhas do conjunto de resultados devem ser retornadas. `n` é um inteiro entre 0 e 4294967295. Se também for especificado `PERCENT`, somente devem ser retornados os primeiros `n` porcentos de linhas do conjunto de resultados. Quando especificado com `PERCENT`, `n` deve ser um inteiro entre 0 e 100. Se a consulta incluir a cláusula `ORDER BY`, as primeiras `n` linhas (ou `n` por cento das linhas) ordenadas pela cláusula `ORDER BY` são retornadas. Se a consulta não possuir a cláusula `ORDER BY`, a ordem das linhas é arbitrária. SQL Server Books Online (N. do T.)
4. Oracle — A cláusula `FOR UPDATE` permite bloquear as linhas selecionadas, não permitindo assim que outros usuários bloqueiem ou atualizem estas linhas até a transação terminar. Esta cláusula somente pode ser especificada no comando `SELECT` de nível mais alto (não em subseleções). Esta cláusula não pode ser especificada junto com as seguintes construções: o operador `DISTINCT`, expressão de `CURSOR`, operadores de conjunto, cláusula `GROUP BY` e funções de agregação. Oracle9i SQL Reference - Release 2 (9.2) - Part Number A96540-02 ([http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/statements\\_103a.htm#2066347](http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/statements_103a.htm#2066347)) (N. do T.)
5. SQL Server — predicado é uma expressão que é avaliada como `TRUE`, `FALSE` ou `UNKNOWN`. Os predicados são usados nas condições de procura das cláusulas `WHERE` e `HAVING`, e nas condições de junção das cláusulas `FROM`. Veja também `BETWEEN`, `CONTAINS`, `EXISTS`, `FREETEXT`, `IN`, `IS [NOT] NULL` e `LIKE`. SQL Server Books Online (N. do T.)
6. No PostgreSQL, no Oracle e no DB2 o valor nulo é classificado em uma posição mais alta que os demais valores, mas no SQL Server o valor nulo é classificado em uma posição mais baixa que os demais valores. O Oracle permite especificar `NULLS FIRST` e `NULLS LAST`, para cada coluna, na cláusula `ORDER BY`. (N. do T.)
7. `collation`; `collating sequence` — Um método para comparar duas cadeias de caracteres comparáveis. Todo conjunto de caracteres possui seu `collation` padrão. (Second Informal Review Draft) ISO/IEC 9075:1992, Database Language SQL- July 30, 1992. (N. do T.)

# SELECT INTO

## Nome

`SELECT INTO` — cria uma tabela a partir dos resultados de uma consulta

## Sinopse

```
SELECT [ ALL | DISTINCT [ ON ( expressão [, ...] ) ] ]  
      * | expressão [ AS nome_de_saída ] [, ...]  
      INTO [ TEMPORARY | TEMP ] [ TABLE ] nova_tabela  
      [ FROM item_do_from [, ...] ]  
      [ WHERE condição ]  
      [ GROUP BY expressão [, ...] ]  
      [ HAVING condição [, ...] ]  
      [ { UNION | INTERSECT | EXCEPT } [ ALL ] seleção ]  
      [ ORDER BY expressão [ ASC | DESC | USING operador ] [, ...] ]  
      [ LIMIT { contador | ALL } ]  
      [ OFFSET início ]  
      [ FOR UPDATE [ OF nome_da_tabela [, ...] ] ]
```

## Descrição

O comando `SELECT INTO` cria uma tabela e a carrega com dados computados por uma consulta. Os dados não são retornados para o cliente, como acontece normalmente no comando `SELECT`. As colunas da nova tabela possuem o mesmo nome e tipo de dado das colunas da saída do comando `SELECT`.

## Parâmetros

`TEMPORARY` ou `TEMP`

Quando especificado, a tabela é criada como uma tabela temporária. Consulte o comando `CREATE TABLE` para obter detalhes.

*nova\_tabela*

O nome (opcionalmente qualificado pelo esquema) da tabela a ser criada.

Todos os outros parâmetros estão descritos detalhadamente no comando `SELECT`.

## Observações

O comando `CREATE TABLE AS` é funcionalmente semelhante ao comando `SELECT INTO`. O comando `CREATE TABLE AS` é a sintaxe recomendada, uma vez que esta forma do comando `SELECT INTO` não está disponível no ECPG nem no PL/pgSQL, porque estes interpretam a cláusula `INTO` de forma diferente. Além disso, o comando `CREATE TABLE AS` oferece um conjunto maior de funcionalidades do que as oferecidas pelo comando `SELECT INTO`.

Antes do PostgreSQL 8.0, a tabela criada pelo comando `SELECT INTO` sempre incluía os OIDs. A partir do PostgreSQL 8.0 a inclusão dos OIDs na tabela criada pelo comando `SELECT INTO` é controlada pelo parâmetro de configuração `default_with_oids`. Atualmente o valor padrão é `TRUE`, mas provavelmente o valor padrão será mudado para `FALSE` em uma versão futura do PostgreSQL.

## Exemplos

Criar a tabela `filmes_recentes` consistindo apenas das entradas recentes na tabela `filmes`:

```
SELECT * INTO filmes_recentes FROM filmes WHERE data_prod >= '2002-01-01';
```

## **Compatibilidade**

O padrão SQL utiliza o comando `SELECT . . . INTO` para representar a seleção de valores colocados dentro de variáveis escalares do programa hospedeiro, em vez de criar uma nova tabela. Esta é a mesma utilização encontrada no ECPG (consulte o Capítulo 29) e no PL/pgSQL (consulte o Capítulo 35). A utilização pelo PostgreSQL do comando `SELECT INTO` para representar a criação de uma tabela é histórica. É melhor utilizar o comando `CREATE TABLE AS` para esta finalidade nos programas novos.

## **Consulte também**

*CREATE TABLE AS*

# SET

## Nome

SET — muda um parâmetro de tempo de execução

## Sinopse

```
SET [ SESSION | LOCAL ] nome { TO | = } { valor | 'valor' | DEFAULT }  
SET [ SESSION | LOCAL ] TIME ZONE { zona_horária | LOCAL | DEFAULT }
```

## Descrição

O comando `SET` muda os parâmetros de configuração de tempo de execução. Muitos parâmetros de tempo de execução listados na Seção 16.4 podem ser mudados em tempo de execução pelo comando `SET` (Mas alguns requerem privilégio de superusuário para serem mudados, e outros não podem ser mudados após o servidor ou a sessão iniciar). O comando `SET` afeta apenas os valores utilizados na sessão corrente.

Se o comando `SET`, ou `SET SESSION`, for executado dentro de uma transação interrompida posteriormente, os efeitos produzidos pelo comando `SET` desaparecem quando a transação é desfeita (Este comportamento representa uma mudança em relação às versões do PostgreSQL anteriores a 7.3, onde os efeitos produzidos pelo comando `SET` não eram desfeitos após um erro posterior). Se a transação que envolve o comando for efetivada, os efeitos produzidos persistem até o fim da sessão, a não ser que seja mudado por outro comando `SET`.

Os efeitos produzidos pelo comando `SET LOCAL` duram apenas até o fim da transação corrente, seja esta efetivada ou não. Um caso especial ocorre quando o comando `SET` é seguido por um comando `SET LOCAL` dentro da mesma transação: o valor do comando `SET LOCAL` tem efeito até o final da transação, mas depois (se a transação for efetivada) passa a ter efeito o valor do comando `SET`.

## Parâmetros

**SESSION**

Especifica que o comando tem efeito para a sessão corrente (Este é o padrão se nem `SESSION` nem `LOCAL` estiverem presentes).

**LOCAL**

Especifica que o comando tem efeito apenas para a transação corrente. Após o `COMMIT` ou o `ROLLBACK` a definição no nível de sessão volta a ter efeito novamente. Deve ser observado que o comando `SET LOCAL` parece não produzir nenhum efeito quando executado fora de um bloco `BEGIN`, uma vez que a transação termina imediatamente.

***nome***

Nome de um parâmetro de tempo de execução cujo valor pode ser definido. Os parâmetros disponíveis estão documentados na Seção 16.4 e abaixo.

***valor***

O novo valor do parâmetro. Os valores podem ser especificados como constantes cadeias de caracteres, identificadores, números, ou listas separadas por vírgula dos mesmos. Pode ser utilizado `DEFAULT` para especificar a redefinição do parâmetro com o seu valor padrão.

Além dos parâmetros de configuração documentados na Seção 16.4, existem uns poucos que somente podem ser ajustados usando o comando `SET`, ou que possuem uma sintaxe especial:

**NAMES**

`SET NAMES valor` é um outro nome para `SET client_encoding TO valor`.

**SEED**

Define a semente interna para o gerador de números randômicos (a função `random`). Os valores permitidos são números de ponto flutuante entre 0 e 1, os quais são então multiplicados por  $2^{31}-1$ .

A semente também pode ser definida chamando a função `setseed`:

```
SELECT setseed(valor);
```

#### TIME ZONE

`SET TIME ZONE valor` é um outro nome para `SET timezone TO valor`. A sintaxe `SET TIME ZONE` permite uma sintaxe especial para especificar a zona horária. Abaixo estão alguns exemplos de valores válidos:

```
'PST8PDT'
```

A zona horária de Berkeley, Califórnia.

```
'Europe/Rome'
```

A zona horária da Itália.

```
-7
```

A zona horária 7 horas a oeste da UTC (equivalente a PDT - Pacific Daylight Time). Os valores positivos estão a leste <sup>1</sup> da UTC.

```
INTERVAL '-08:00' HOUR TO MINUTE
```

A zona horária 8 horas a oeste da UTC (o mesmo que PST - Pacific Standard Time = UTC - 8 horas).

```
LOCAL
```

```
DEFAULT
```

Define a zona horária como a zona horária local (a zona horária padrão do sistema operacional do servidor).

Consulte a Seção 8.5 para obter informações adicionais sobre zonas horárias. Além disso, o Apêndice B possui uma lista de nomes reconhecidos de zonas horárias.

## Observações

A função `set_config` fornece uma funcionalidade equivalente. Consulte a Seção 9.20.

## Exemplos

Definir o caminho de procura de esquema:

```
SET search_path TO meu_esquema, public;
```

Definir o estilo da data igual ao estilo tradicional do POSTGRES, com a convenção de entrada “dia antes do mês”:

```
SET datestyle TO postgres, dmy;
```

Definir a zona horária de Berkeley, California:

```
SET TIME ZONE 'PST8PDT';
```

Definir a zona horária da Itália:

```
SET TIME ZONE 'Europe/Rome';
```

## Compatibilidade

`SET TIME ZONE` estende a sintaxe definida no padrão SQL. O padrão permite somente deslocamentos de zona horária numéricos, enquanto o PostgreSQL permite uma especificação de zona horária mais flexível. Todas as outras funcionalidades do SET são extensões do PostgreSQL.

## Consulte também

*RESET, SHOW*

**Notas**

1. Leste = oriente, o lado do Sol nascente; Oeste = ocidente, o lado do Sol poente. (N. do T.)

# SET CONSTRAINTS

## Nome

`SET CONSTRAINTS` — define os modos de verificação da restrição na transação corrente

## Sinopse

```
SET CONSTRAINTS { ALL | nome [, ...] } { DEFERRED | IMMEDIATE }
```

## Descrição

O comando `SET CONSTRAINTS` define o comportamento da verificação da restrição <sup>1 2</sup> dentro da transação corrente. No modo `IMMEDIATE` (imediato), as restrições são verificadas ao final de cada comando. No modo `DEFERRED` (postergado), as restrições não são verificadas até a transação ser efetivada (`commit`). Cada restrição possui seu próprio modo `IMMEDIATE` ou `DEFERRED`.

Ao ser criada, a restrição sempre recebe uma destas três características: `DEFERRABLE INITIALLY DEFERRED` (postergável, inicialmente postergada), `DEFERRABLE INITIALLY IMMEDIATE` (postergável, inicialmente imediata), ou `NOT DEFERRABLE` (não postergável). A terceira classe é sempre `IMMEDIATE` (imediata) e não é afetada pelo comando `SET CONSTRAINTS`. As duas primeiras classes começam todas as transações no modo indicado, mas seus comportamentos podem ser modificados dentro da transação pelo comando `SET CONSTRAINTS`.

O comando `SET CONSTRAINTS` com uma lista de nomes de restrição muda o modo destas restrições apenas (que devem ser todos postergáveis). Se existirem várias restrições correspondendo a um nome fornecido, todos eles são afetados. O comando `SET CONSTRAINTS ALL` muda o modo de todas as restrições postergáveis.

Quando o comando `SET CONSTRAINTS` muda o modo da restrição de `DEFERRED` para `IMMEDIATE`, o novo modo entra em vigor retroativamente: toda modificação de dados pendente, que deveria ser verificada no final da transação, é verificada durante a execução do comando `SET CONSTRAINTS`. Se alguma destas restrições for violada, o comando `SET CONSTRAINTS` falha (e não muda o modo da restrição). Portanto, o comando `SET CONSTRAINTS` pode ser utilizado para obrigar que ocorra a verificação das restrições em um determinado ponto da transação.

Atualmente, somente as restrições de chave estrangeira são afetadas por esta definição. As restrições de verificação (`check`) e de unicidade são sempre, efetivamente, não postergáveis.

## Observações

Este comando somente altera o comportamento das restrições dentro da transação corrente. Portanto, se este comando for executado fora de um bloco de transação (par `BEGIN/COMMIT`), parecerá que não produziu nenhum efeito.

## Compatibilidade

Este comando está em conformidade com o comportamento definido no padrão SQL, exceto pela limitação que, no PostgreSQL, somente se aplica às restrições de chave estrangeira.

O padrão SQL diz que os nomes das restrições que aparecem no comando `SET CONSTRAINTS` podem ser qualificados pelo esquema. Esta funcionalidade ainda não é suportada pelo PostgreSQL: os nomes não podem ser qualificados, e todas as restrições correspondendo ao comando serão afetadas, não importando o esquema em que estejam.

## Notas

1. As restrições de integridade, geralmente chamadas apenas de restrições, definem os estados válidos dos dados SQL restringindo os valores nas tabelas base. Uma restrição pode ser uma restrição de tabela, uma restrição de domínio ou uma asserção. (Second Informal Review Draft) ISO/IEC 9075:1992, Database Language SQL- July 30, 1992 (N. do T.)
2. Oracle — O comando `SET CONSTRAINTS` é utilizado para especificar, para uma determinada transação, se a restrição postergável será verificada após cada comando da DML, ou quando a transação for efetivada. `SET CONSTRAINT[S]` ([http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/statements\\_104a.htm](http://www.stanford.edu/dept/itss/docs/oracle/9i/server.920/a96540/statements_104a.htm)) (N. do T.)

# SET SESSION AUTHORIZATION

## Nome

`SET SESSION AUTHORIZATION` — define o identificador do usuário da sessão e o identificador do usuário corrente, da sessão corrente.

## Sinopse

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION nome_do_usuario
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

## Descrição

Este comando define o identificador do usuário da sessão, e o identificador do usuário corrente, no contexto da sessão SQL corrente, como sendo *nome\_do\_usuario*. O nome do usuário pode ser escrito tanto como um identificador quanto como um literal cadeia de caracteres. Usando este comando é possível, por exemplo, se tornar temporariamente um usuário sem privilégios e posteriormente voltar a ser um superusuário.

O identificador do usuário da sessão é inicialmente definido como sendo o (possivelmente autenticado) nome do usuário fornecido pelo cliente. O identificador do usuário corrente normalmente é igual ao identificador do usuário da sessão, mas pode mudar temporariamente no contexto das funções “setuid” e de outros mecanismos semelhantes. O identificador do usuário corrente é relevante para verificar as permissões.

O identificador do usuário da sessão somente pode ser mudado se o usuário inicial da sessão (o *usuário autenticado*) possuir o privilégio de superusuário. Senão, o comando é aceito somente se especificar o nome do usuário autenticado.

Os modificadores `SESSION` e `LOCAL` atuam da mesma forma que atuam no comando *SET* comum.

As formas `DEFAULT` e `RESET` redefinem os identificadores de usuário da sessão e corrente como sendo o nome do usuário autenticado originalmente. Estas formas são sempre aceitas.

## Exemplos

```
SELECT SESSION_USER, CURRENT_USER;
```

session_user	current_user
pedro	pedro

```
SET SESSION AUTHORIZATION 'paulo';
```

```
SELECT SESSION_USER, CURRENT_USER;
```

session_user	current_user
paulo	paulo

## Compatibilidade

O padrão SQL permite algumas outras expressões aparecerem no lugar do literal *nome\_do\_usuario*, as quais não são importantes na prática. O PostgreSQL permite a sintaxe de identificador ("*nome\_do\_usuario*"), que o SQL não permite. O padrão SQL não permite este comando durante uma transação; O PostgreSQL não faz esta restrição, porque não há razão para fazê-la. O padrão deixa os privilégios necessários para executar este comando por conta da implementação.



# SET TRANSACTION

## Nome

`SET TRANSACTION` — define as características da transação corrente

## Sinopse

```
SET TRANSACTION modo_da_transação [, ...]
```

```
SET SESSION CHARACTERISTICS AS TRANSACTION modo_da_transação [, ...]
```

onde *modo\_da\_transação* é um entre:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }  
READ WRITE | READ ONLY
```

## Descrição

O comando `SET TRANSACTION` define as características da transação corrente. Não produz nenhum efeito para as transações subseqüentes. O comando `SET SESSION CHARACTERISTICS` define o padrão das características das transações para as próximas transações na sessão. Estes padrões podem ser mudados pelo comando `SET TRANSACTION` para uma transação individual.

As características de transação disponíveis são o nível de isolamento da transação e o modo de acesso da transação (ler/escrever ou somente para leitura).

O nível de isolamento de uma transação determina quais dados a transação pode ver quando outras transações estão processando concorrentemente.

`READ COMMITTED`

A declaração consegue ver apenas as linhas efetivadas (`commit`) antes do início da sua execução. Este é o padrão.

`SERIALIZABLE`

Todos os comandos da transação corrente podem ver apenas as linhas efetivadas antes da primeira consulta ou comando de modificação de dados ter sido executado nesta transação.

O padrão SQL define dois níveis adicionais, `READ UNCOMMITTED` e `REPEATABLE READ`. No PostgreSQL `READ UNCOMMITTED` é tratado como `READ COMMITTED`, enquanto `REPEATABLE READ` é tratado como `SERIALIZABLE`.

O nível de isolamento da transação não pode ser mudado após a primeira consulta ou comando de modificação de dado da transação (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH` ou `COPY`) ter sido executado. Consulte o Capítulo 12 para obter informações adicionais sobre o isolamento de transações e controle de simultaneidade.

O modo de acesso da transação determina se a transação lê e escreve, ou se é somente para leitura. Ler e escrever é o padrão. Quando a transação é somente para leitura, os seguintes comandos SQL não são permitidos: `INSERT`, `UPDATE`, `DELETE` e `COPY` se a tabela a ser escrita não for uma tabela temporária; todos os comandos `CREATE`, `ALTER` e `DROP`; `COMMENT`, `GRANT`, `REVOKE`, `TRUNCATE`; também `EXPLAIN ANALYZE` e `EXECUTE` se o comando a ser executado está entre os listados. Esta é uma noção de somente para leitura de alto nível, que não impede escritas no disco.

## Observações

Se for executado o comando `SET TRANSACTION` sem ser executado antes o comando `START TRANSACTION` ou `BEGIN`, vai ficar parecendo que este não causou nenhum efeito, uma vez que a transação termina imediatamente.

É possível não utilizar o comando `SET TRANSACTION`, especificando o *modo\_da\_transação* desejado no `BEGIN` ou no `START TRANSACTION`.

Os modos de transação padrão da sessão também podem ser definidos através dos parâmetros de configuração `default_transaction_isolation` e `default_transaction_read_only` (De fato, `SET SESSION CHARACTERISTICS` é apenas uma forma verbosa equivalente a definir estas variáveis com `SET`). Isto significa que os valores padrão podem ser definidos no arquivo de configuração, via `ALTER DATABASE`, etc. Consulte a Seção 16.4 para obter informações adicionais.

## Compatibilidade

Os dois comandos estão definidos no padrão SQL. No padrão SQL `SERIALIZABLE` é o nível de isolamento padrão; no PostgreSQL normalmente o padrão é `READ COMMITTED`, mas pode ser mudado conforme mencionado acima. Devido à falta de bloqueio de predicado, o nível `SERIALIZABLE` não é verdadeiramente serializável. Consulte o Capítulo 12 para obter detalhes.

No padrão SQL existe uma outra característica de transação que pode ser definida por estes comandos: o tamanho da área de diagnósticos. Este conceito só se aplica à linguagem SQL incorporada e, portanto, não é implementado no servidor PostgreSQL.

O padrão SQL requer a presença de vírgulas entre os *modo\_da\_transação* sucessivos, mas por razões históricas o PostgreSQL permite a omissão destas vírgulas.

# SHOW

## Nome

SHOW — mostra o valor de um parâmetro de tempo de execução

## Sinopse

```
SHOW nome
SHOW ALL
```

## Descrição

O comando `SHOW` mostra a definição corrente de parâmetros de tempo de execução. Estas variáveis podem ser definidas utilizando o comando `SET`, editando o arquivo de configuração `postgresql.conf`, pela variável de ambiente `PGOPTIONS` (ao se usar a `libpq` ou aplicativos baseados na `libpq`), ou por meio de sinalizadores de linha de comando ao iniciar o `postmaster`. Consulte a Seção 16.4 para obter informações adicionais.

## Parâmetros

*nome*

O nome de um parâmetro de tempo de execução. Os parâmetros disponíveis estão documentados na Seção 16.4 e na página de referência do comando `SET`. Além disso, existem uns poucos parâmetros que podem ser mostrados mas não podem ser definidos:

`SERVER_VERSION`

Mostra o número da versão do servidor.

`SERVER_ENCODING`

Mostra o conjunto de codificação de caracteres do lado servidor. Atualmente, este parâmetro pode ser mostrado mas não pode ser definido, porque a codificação é definida na hora da criação do banco de dados.

`LC_COLLATE`

Mostra a definição de idioma do banco de dados para agrupamento (`collation`<sup>1</sup>) (ordenação do texto). Atualmente, este parâmetro pode ser mostrado mas não pode ser definido, porque a definição é determinada pelo comando `initdb`.

`LC_CTYPE`

Mostra a definição do idioma do banco de dados para classificação de caracteres.<sup>2</sup> Atualmente, este parâmetro pode ser mostrado mas não pode ser definido, porque a definição é determinada pelo comando `initdb`.

`IS_SUPERUSER`

Verdade se o identificador de autorização da sessão corrente tiver privilégios de superusuário.

`ALL`

Mostra os valores de todos os parâmetros de configuração.

## Observações

A função `current_setting` produz uma saída equivalente. Consulte a Seção 9.20.

## Exemplos

Mostrar a definição corrente do parâmetro `DateStyle` (estilo da data):

```
SHOW DateStyle;
```

```
DateStyle
-----
ISO, MDY
(1 linha)
```

Mostrar a definição corrente do parâmetro `geqo`:

```
SHOW geqo;
```

```
geqo
-----
on
(1 linha)
```

Mostrar todas as definições:

```
SHOW ALL;
```

name	setting
add_missing_from	on
archive_command	unset
australian_timezones	off
.	
.	
.	
work_mem	1024
zero_damaged_pages	off

(140 linhas)

## Compatibilidade

O comando `SHOW` é uma extensão do PostgreSQL.

## Consulte também

*SET*, *RESET*

## Notas

1. `collation; collating sequence` — Um método para comparar duas cadeias de caracteres comparáveis. Todo conjunto de caracteres possui seu `collation` padrão. (Second Informal Review Draft) ISO/IEC 9075:1992, Database Language SQL- July 30, 1992. (N. do T.)
2. Define a classificação do caractere, conversão maiúscula/minúscula, e outros atributos do caractere. `LC_CTYPE` Category for the Locale Definition Source File Format ([http://publibn.boulder.ibm.com/doc\\_link/en\\_US/a\\_doc\\_lib/files/aixfiles/LC\\_CTYPE.htm](http://publibn.boulder.ibm.com/doc_link/en_US/a_doc_lib/files/aixfiles/LC_CTYPE.htm)) (N. do T.)

# START TRANSACTION

## Nome

`START TRANSACTION` — inicia um bloco de transação

## Sinopse

```
START TRANSACTION [ modo_da_transação [, ...] ]
```

onde *modo\_da\_transação* é um entre:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }  
READ WRITE | READ ONLY
```

## Descrição

Este comando inicia um novo bloco transação. Se for especificado o nível de isolamento, ou modo de ler/escrever, a nova transação terá estas características, como se *SET TRANSACTION* fosse executado. É idêntico ao comando *BEGIN*.

## Parâmetros

Veja em *SET TRANSACTION* as informações sobre o significado dos parâmetros deste comando.

## Compatibilidade

No padrão, não é necessário utilizar `START TRANSACTION` para iniciar um bloco de transação: qualquer comando SQL inicia, implicitamente, um bloco de transação. O comportamento do PostgreSQL pode ser visto como utilizando, implicitamente, um comando `COMMIT` após cada comando que não vem após um comando `START TRANSACTION` (ou `BEGIN`) sendo isto, portanto, geralmente chamado de “auto-efetivação” (`autocommit`). Outros sistemas de bancos de dados relacionais podem oferecer a funcionalidade de auto-efetivação como uma comodidade.

O padrão SQL requer a presença de vírgulas entre os *modos\_da\_transação*, mas por razões históricas o PostgreSQL permite a omissão destas vírgulas.

Consulte também a seção sobre compatibilidade de *SET TRANSACTION*.

## Consulte também

*BEGIN*, *COMMIT*, *ROLLBACK*, *SAVEPOINT*, *SET TRANSACTION*

# TRUNCATE

## Nome

TRUNCATE — esvazia a tabela

## Sinopse

```
TRUNCATE [ TABLE ] nome
```

## Descrição

O comando TRUNCATE remove rapidamente todas as linhas da tabela. Produz o mesmo efeito do comando DELETE não qualificado (sem WHERE), mas como na verdade não varre a tabela é mais rápido. É mais útil em tabelas grandes.

## Parâmetros

*nome*

O nome (opcionalmente qualificado pelo esquema) da tabela a ser truncada.

## Observações

O comando TRUNCATE não pode ser usado quando existe referência de chave estrangeira de outra tabela para a tabela. Neste caso a verificação da validade tornaria necessária a varredura da tabela, e o ponto central é não fazê-la.

O comando TRUNCATE não executa nenhum gatilho ON DELETE definido pelo usuário, porventura existente na tabela.

## Exemplos

Truncar a tabela `tbl_grande`:

```
TRUNCATE TABLE tbl_grande;
```

## Compatibilidade

Não existe o comando TRUNCATE no padrão SQL.

# UNLISTEN

## Nome

UNLISTEN — pára de ouvir uma notificação

## Sinopse

```
UNLISTEN { nome | * }
```

## Descrição

O comando UNLISTEN é utilizado para remover um registro para eventos de NOTIFY existente. O comando UNLISTEN cancela o registro existente da sessão corrente do PostgreSQL como ouvinte da notificação *nome*. O curinga especial \* cancela todos os registros de ouvinte da sessão corrente.

O comando *NOTIFY* contém uma explicação mais extensa sobre a utilização dos comandos LISTEN e NOTIFY.

## Parâmetros

*nome*

O nome da notificação (qualquer identificador).

\*

Todos os registros de ouvinte para esta sessão são limpos.

## Observações

É possível deixar de ouvir algo que não estava sendo ouvido; não é mostrado nenhum erro ou advertência.

Ao final de cada sessão, o comando UNLISTEN \* é executado automaticamente.

## Exemplos

Para registrar:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

Após o comando UNLISTEN ter sido executado, os comandos NOTIFY posteriores são ignorados:

```
UNLISTEN virtual;
NOTIFY virtual;
-- não é recebido nenhum evento NOTIFY
```

## Compatibilidade

Não existe o comando UNLISTEN no padrão SQL.

## Consulte também

*LISTEN*, *NOTIFY*

# UPDATE

## Nome

UPDATE — atualiza linhas de uma tabela

## Sinopse

```
UPDATE [ ONLY ] tabela SET coluna = { expressão | DEFAULT } [, ...]  
    [ FROM lista_do_from ]  
    [ WHERE condição ]
```

## Descrição

O comando `UPDATE` muda os valores das colunas especificadas em todas as linhas que satisfazem a condição. Somente as colunas a serem modificadas precisam ser mencionadas na cláusula `SET`; as colunas que não são modificadas explicitamente mantêm seus valores anteriores.

Por padrão, o comando `UPDATE` atualiza linhas na tabela especificada e nas suas tabelas descendentes. Se for desejado atualizar apenas a tabela especificada, deve ser utilizada a cláusula `ONLY`.

Existem duas maneiras de modificar uma tabela utilizando informações contidas em outras tabelas do banco de dados: usando subseleções, ou especificando tabelas adicionais na cláusula `FROM`. A técnica mais apropriada depende das circunstâncias específicas.

É necessário possuir o privilégio `UPDATE` na tabela para atualizá-la, assim como o privilégio `SELECT` em todas as tabelas cujos valores são lidos pela *expressão* ou pela *condição*.

## Parâmetros

*tabela*

O nome (opcionalmente qualificado pelo esquema) da tabela a ser atualizada.

*coluna*

O nome de uma coluna da *tabela*. O nome da coluna pode ser qualificado pelo nome de um subcampo ou índice de matriz, se for necessário.

*expressão*

Uma expressão a ser atribuída à coluna. A expressão pode usar o valor antigo desta e de outras colunas da tabela.

DEFAULT

Define o valor da coluna como o seu valor padrão (que é nulo se nenhuma expressão padrão específica tiver sido atribuída à coluna).

*lista\_do\_from*

Uma lista de expressões de tabela, que permite aparecerem colunas de outras tabelas na condição `WHERE` e nas expressões de atualização. É semelhante à lista de tabelas que pode ser especificada na cláusula *Cláusula FROM* do comando `SELECT`. Deve ser observado que a tabela de destino não deve aparecer na *lista\_do\_from*, a menos que se deseje uma autojunção (na qual deve aparecer com um aliás na *lista\_do\_from*).

*condição*

Uma expressão que retorna um valor do tipo `boolean`. Somente são atualizadas as linhas para as quais esta expressão retorna `true`.

## Saídas

Ao terminar bem-sucedido, o comando `UPDATE` retorna uma linha de fim de comando na forma

```
UPDATE contador
```



O *contador* é o número de linhas atualizadas. Se *contador* for 0, nenhuma linha corresponde à *condição* (o que não é considerado um erro).

## Observações

Quando a cláusula `FROM` está presente o que acontece, essencialmente, é que é feita a junção da tabela de destino com as tabelas mencionadas na *lista\_do\_from*, e cada linha produzida pela junção representa uma operação de atualização para a tabela de destino. Ao usar a cláusula `FROM` deve-se garantir que a junção produz, no máximo, uma linha de saída para cada linha a ser modificada. Em outras palavras, a linha de destino não deve ser juntada com mais de uma linha das outras tabelas, porque senão somente uma das linhas da junção será utilizada para atualizar a linha de destino, mas qual delas será utilizada não é prontamente previsível.

Devido a esta indeterminação, é mais seguro fazer referência a outras tabelas somente dentro de subseleções, embora seja mais difícil de ler e mais lento do que a utilização da junção.

## Exemplos

Mudar a palavra `Drama` para `Dramático` na coluna `tipo` da tabela `filmes`:

```
UPDATE filmes SET tipo = 'Dramático' WHERE tipo = 'Drama';
```

Ajustar as entradas de temperatura e redefinir a precipitação com seu valor padrão em uma linha da tabela `clima`:

```
UPDATE clima SET temp_min = temp_min+1, temp_max = temp_min+15, precipitacao = DEFAULT
WHERE cidade = 'São Francisco' AND data = '2003-07-03';
```

Incrementar o contador de vendas do vendedor que gerencia a conta da Corporação Acme, usando a sintaxe da cláusula `FROM`:

```
UPDATE empregados SET contador_de_vendas = contador_de_vendas + 1 FROM contas
WHERE contas.nome = 'Corporação Acme'
AND empregados.id = contas.vendedor;
```

Realizar a mesma operação utilizando uma subseleção na cláusula `WHERE`:

```
UPDATE empregados SET contador_de_vendas = contador_de_vendas + 1 WHERE id =
(SELECT vendedor FROM contas WHERE nome = 'Corporação Acme');
```

Tentar inserir um novo item no estoque junto com a quantidade em estoque. Se o item já existir, em vez inserir, atualizar o contador do item em estoque. Para fazer isto sem que a transação falhe é utilizado ponto de salvamento.

```
BEGIN;
-- outras operações
SAVEPOINT sp1;
INSERT INTO vinhos VALUES('Chateau Lafite 2003', '24');
-- Assumindo que o comando acima falhe devido à violação de chave única,
-- serão executados os comandos abaixo:
ROLLBACK TO sp1;
UPDATE vinhos SET estoque = estoque + 24 WHERE nome_do_vinho = 'Chateau Lafite 2003';
-- prosseguir com a outras operações e, eventualmente,
COMMIT;
```

## Compatibilidade

Este comando está em conformidade com o padrão SQL, exceto pela cláusula `FROM` que é uma extensão do PostgreSQL.

Alguns outros sistemas de bancos de dados oferecem uma opção `FROM`, onde a tabela de destino é supostamente listada novamente dentro da cláusula `FROM`. Não é desta forma que o PostgreSQL interpreta a cláusula `FROM`. Deve-se tomar cuidado ao portar aplicativos que utilizam esta extensão.

# VACUUM

## Nome

VACUUM — limpa e opcionalmente analisa um banco de dados

## Sinopse

```
VACUUM [ FULL | FREEZE ] [ VERBOSE ] [ tabela ]  
VACUUM [ FULL | FREEZE ] [ VERBOSE ] ANALYZE [ tabela [ (coluna [, ...] ) ] ]
```

## Descrição

O comando `VACUUM` recupera a área de armazenamento ocupada pelas tuplas excluídas. Na operação normal do PostgreSQL as tuplas excluídas, ou tornadas obsoletas por causa de uma atualização, não são fisicamente removidas da tabela; permanecem presentes até o comando `VACUUM` ser executado. Portanto, é necessário executar o comando `VACUUM` periodicamente, especialmente em tabelas freqüentemente atualizadas.

Sem nenhum parâmetro, o comando `VACUUM` processa todas as tabelas do banco de dados corrente. Com um parâmetro, o `VACUUM` processa somente esta tabela.

O comando `VACUUM ANALYZE` executa o comando `VACUUM` seguido pelo comando `ANALYZE` para cada tabela selecionada. Esta é uma forma de combinação útil para scripts de rotinas de manutenção. Consulte o comando `ANALYZE` para obter mais detalhes sobre o seu processamento.

O comando `VACUUM` simples (sem o `FULL`) apenas recupera o espaço, tornando-o disponível para ser reutilizado. Esta forma do comando pode operar em paralelo com a leitura e escrita normal, porque não requer o bloqueio exclusivo da tabela. O `VACUUM FULL` executa um processamento mais extenso, incluindo a movimentação das tuplas entre blocos para tentar compactar a tabela no menor número de blocos de disco possível. Esta forma é muito mais lenta, e requer o bloqueio exclusivo da tabela para processá-la.

O `FREEZE` é uma opção com finalidade especial, que faz as tuplas serem marcadas como “congeladas” logo que possível, em vez de aguardar até estarem inteiramente antigas. Se for feito quando não houver nenhuma outra transação aberta no mesmo banco de dados, então é garantido que todas as tuplas do banco de dados sejam “congeladas”, não ficando mais sujeitas aos problemas do recomeço do ID de transação, não importando quanto tempo o banco de dados seja deixado sem o comando `VACUUM` ser executado. O `FREEZE` não é recomendado para uso rotineiro. Sua única utilização pretendida é em conjunto com a preparação de bancos de dados modelo definido pelo usuário, ou outros bancos de dados inteiramente somente para leitura, que não receberão operações `VACUUM` na rotina de manutenção. Consulte o Capítulo 21 para obter detalhes.

## Parâmetros

`FULL`

Seleciona uma limpeza “completa”, que pode recuperar mais espaço, mas é muito mais demorada e bloqueia a tabela em modo exclusivo.

`FREEZE`

Seleciona um “congelamento” agressivo das tuplas.

`VERBOSE`

Mostra, para cada tabela, um relatório da atividade de limpeza detalhado.

`ANALYZE`

Atualiza as estatísticas utilizadas pelo planejador para determinar o modo mais eficiente de executar um comando.

*tabela*

O nome (opcionalmente qualificado pelo esquema) da tabela específica a ser limpa. Por padrão todas as tabelas do banco de dados corrente.

*coluna*

O nome da coluna especifica a ser analisada. Por padrão todas as colunas.

## Saídas

Quando é especificado `VERBOSE`, o comando `VACUUM` mostra mensagens de progresso indicando qual tabela está sendo processada no momento. Também são mostradas várias estatísticas sobre as tabelas.

## Observações

Recomenda-se que os bancos de dados de produção ativos sejam limpos com frequência (pelo menos toda noite), para remover as linhas expiradas. Após adicionar ou remover um grande número de linhas, aconselha-se executar o comando `VACUUM ANALYZE` para a tabela afetada. Este procedimento atualizará os catálogos do sistema com os resultados de todas as mudanças recentes, permitindo o planejador de comandos do PostgreSQL fazer melhores escolhas ao planejar os comandos.

A opção `FULL` não é recomendada para uso rotineiro, mas pode ser útil em casos especiais. Um exemplo é após ter sido excluída a maioria das linhas da tabela e deseja-se que a tabela seja fisicamente encolhida para ocupar menos espaço em disco. O comando `VACUUM FULL` geralmente encolhe mais a tabela que o comando `VACUUM` simples.

## Exemplos

Abaixo está mostrado um exemplo da execução do comando `VACUUM` em uma tabela do banco de dados `regression`:

```
regression=# VACUUM VERBOSE ANALYZE onek;
INFO:  vacuuming "public.onek"
INFO:  index "onek_unique1" now contains 1000 tuples in 14 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.18 sec.
INFO:  index "onek_unique2" now contains 1000 tuples in 16 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.07u sec elapsed 0.23 sec.
INFO:  index "onek_hundred" now contains 1000 tuples in 13 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.17 sec.
INFO:  index "onek_stringul" now contains 1000 tuples in 48 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.09u sec elapsed 0.59 sec.
INFO:  "onek": removed 3000 tuples in 108 pages
DETAIL:  CPU 0.01s/0.06u sec elapsed 0.07 sec.
INFO:  "onek": found 3000 removable, 1000 nonremovable tuples in 143 pages
DETAIL:  0 dead tuples cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 0.07s/0.39u sec elapsed 1.56 sec.
INFO:  analyzing "public.onek"
INFO:  "onek": 36 pages, 1000 rows sampled, 1000 estimated total rows
VACUUM
```

## Compatibilidade

Não existe o comando `VACUUM` no padrão SQL.

## Consulte também

*vacuumdb*

## II. Aplicativos cliente do PostgreSQL

Esta parte contém informações de referência para os aplicativos e utilitários clientes do PostgreSQL. Nem todos estes comandos são de uso geral, alguns requerem privilégios especiais. A característica comum destes aplicativos é poderem ser executados a partir de qualquer computador, independentemente de onde o servidor de banco de dados está instalado.

# clusterdb

## Nome

clusterdb — agrupa um banco de dados do PostgreSQL

## Sinopse

```
clusterdb [opção_de_conexão...] [--table | -t table ] [nome_do_banco_de_dados]  
clusterdb [opção_de_conexão...] [--all | -a]
```

## Descrição

O clusterdb é um utilitário para reagrupar tabelas em um banco de dados do PostgreSQL. Encontra as tabelas que foram agrupadas anteriormente, reagrupando-as novamente utilizando o mesmo índice usado da última vez. As tabelas que nunca foram agrupadas não são afetadas.

O clusterdb é uma capa em torno do comando *CLUSTER* do SQL. Não existe diferença efetiva entre agrupar bancos de dado através deste utilitário, ou através de outros métodos para acessar ao servidor.

## Opções

O clusterdb aceita os seguintes argumentos de linha de comando:

```
-a  
--all
```

Agrupa todos os bancos de dados.

```
[-d] nome_do_banco_de_dados  
[--dbname] nome_do_banco_de_dados
```

Especifica o nome do banco de dados a ser agrupado. Se não for especificado e não for utilizado *-a* (ou *--all*), o nome do banco de dados é lido da variável de ambiente *PGDATABASE*. Se esta variável não estiver definida, então é utilizado o nome do usuário especificado na conexão.

```
-e  
--echo
```

Mostra os comandos que o clusterdb gera e envia para o servidor.

```
-q  
--quiet
```

Não exibe resposta.

```
-t tabela  
--table tabela
```

Agrupa somente a *tabela*.

O clusterdb também aceita os seguintes argumentos de linha de comando para os parâmetros de conexão:

```
-h hospedeiro  
--host hospedeiro
```

Especifica o nome de hospedeiro da máquina onde o servidor está executando. Se o nome iniciar por barra (/) é usado como o diretório do soquete do domínio Unix.

```
-p porta  
--port porta
```

Especifica a porta TCP, ou a extensão do arquivo de soquete do domínio Unix local, onde o servidor está atendendo as conexões.

```
-U nome_do_usuario
--username nome_do_usuario
```

Nome do usuário para conectar.

```
-W
--password
```

Força a solicitação da senha.

## Ambiente

```
PGDATABASE
PGHOST
PGPORT
PGUSER
```

Parâmetros de conexão padrão.

## Diagnósticos

Encontrando dificuldades consulte em *CLUSTER* e *psql* a discussão dos problemas possíveis e as mensagens de erro. O servidor de banco de dados deve estar executando no hospedeiro de destino. Também se aplicam todas as definições de conexão padrão e as variáveis de ambiente utilizadas pela biblioteca cliente libpq.

## Exemplos

Para agrupar o banco de dados teste:

```
$ clusterdb teste
```

Para agrupar apenas a tabela foo no banco de dados chamado xyzzy:

```
$ clusterdb --table foo xyzzy
```

## Consulte também

*CLUSTER*

# createdb

## Nome

`createdb` — cria um novo banco de dados do PostgreSQL

## Sinopse

`createdb` [*opção...*] [*nome\_do\_banco\_de\_dados*] [*descrição*]

## Descrição

O `createdb` cria um banco de dados no PostgreSQL.

Normalmente, o usuário do banco de dados que executa este comando se torna o dono do novo banco de dados. Entretanto, pode ser especificado um dono diferente por meio da opção `-O`, se o usuário que está executando este comando tiver os privilégios apropriados.

O `createdb` é uma capa em torno do comando `CREATE DATABASE` do SQL. Não existe diferença efetiva entre criar bancos de dados através deste utilitário, ou através de outros métodos para acessar o servidor.

## Opções

O `createdb` aceita os seguintes argumentos de linha de comando:

*nome\_do\_banco\_de\_dados*

Especifica o nome do banco de dados a ser criado. O nome deve ser único entre todos os bancos de dados do PostgreSQL deste agrupamento. O padrão é criar o banco de dados com o mesmo nome do usuário atual do sistema operacional.

*descrição*

Especifica um comentário a ser associado ao banco de dados recém criado.

`-D espaço_de_tabelas`

`--tablespace espaço_de_tabelas`

Especifica o espaço de tabelas padrão para o banco de dados.

`-e`

`--echo`

Mostra os comandos que o `createdb` gera e envia para o servidor.

`-E codificação`

`--encoding codificação`

Especifica o esquema de codificação de caracteres a ser usado neste banco de dados. Os conjuntos de caracteres suportados pelo servidor PostgreSQL estão descritos na Seção 20.2.1.

`-O dono`

`--owner dono`

Especifica o usuário do banco de dados que será o dono do novo banco de dados.

`-q`

`--quiet`

Não exibe resposta.

`-T modelo`

`--template modelo`

Especifica o banco de dados modelo, a partir do qual este banco de dados será construído.

As opções `-D`, `-E`, `-O` e `-T` correspondem às opções do comando *CREATE DATABASE* subjacente; consulte este comando para obter informações adicionais sobre estas opções.

O `createdb` também aceita os seguintes argumentos de linha de comando para os parâmetros de conexão:

```
-h hospedeiro
--host hospedeiro
```

Especifica o nome de hospedeiro da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é usado como o diretório do soquete do domínio Unix.

```
-p porta
--port porta
```

Especifica a porta TCP, ou a extensão de arquivo do soquete do domínio Unix local, onde o servidor está atendendo as conexões.

```
-U nome_do_usuario
--username nome_do_usuario
```

Nome do usuário para conectar.

```
-W
--password
```

Força a solicitação da senha.

## Ambiente

`PGDATABASE`

Se estiver definido, o nome do banco de dados a ser criado, a menos que o nome esteja definido na linha de comando.

`PGHOST`

`PGPORT`

`PGUSER`

Parâmetros de conexão padrão. `PGUSER` também determina o nome do banco de dados a ser criado, se este não for especificado na linha de comando ou por `PGDATABASE`.

## Diagnósticos

Havendo dificuldade, veja no comando *CREATE DATABASE* e no `psql` a discussão dos problemas possíveis e as mensagens de erro. O servidor de banco de dados deve estar executando no hospedeiro de destino. Também se aplicam todas as definições de conexão padrão e as variáveis de ambiente utilizadas pela biblioteca cliente `libpq`.

## Exemplos

Para criar o banco de dados `demo` usando o servidor de banco de dados padrão:

```
$ createdb demo
CREATE DATABASE
```

A resposta é a mesma que teria sido recebida se fosse executado o comando `CREATE DATABASE` do SQL.

Para criar o banco de dados `demo` usando o servidor no hospedeiro `eden`, a porta 5000, o esquema de codificação `LATIN1` e vendo o comando subjacente:

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE "demo" WITH ENCODING = 'LATIN1'
CREATE DATABASE
```

## Consulte também

`dropdb`, *CREATE DATABASE*



# createlang

## Nome

createlang — cria uma linguagem procedural do PostgreSQL

## Sinopse

```
createlang [opção_de_conexão...] nome_da_linguagem [nome_do_banco_de_dados]
createlang [opção_de_conexão...] --list | -l nome_do_banco_de_dados
```

## Descrição

O createlang é um utilitário para adicionar uma nova linguagem de programação a um banco de dados do PostgreSQL. O createlang pode tratar todas as linguagens fornecidas na distribuição padrão do PostgreSQL, mas não as linguagens fornecidas por terceiros.

Embora as linguagens de programação do servidor possam ser adicionadas diretamente usando vários comandos SQL, recomenda-se o uso do aplicativo createlang, porque este realiza várias verificações, e é muito mais fácil usá-lo. Consulte o comando *CREATE LANGUAGE* para obter informações adicionais.

## Opções

O createlang aceita os seguintes argumentos de linha de comando:

*nome\_da\_linguagem*

Especifica o nome da linguagem de programação procedural a ser definida.

`[-d] nome_do_banco_de_dados`

`[--dbname] nome_do_banco_de_dados`

Especifica em qual banco de dados a linguagem deve ser adicionada. O padrão é usar o banco de dados com o mesmo nome do usuário corrente do sistema operacional.

`-e`

`--echo`

Mostra os comandos SQL à medida que são executados.

`-l`

`--list`

Mostra a relação de linguagens instaladas no banco de dados de destino.

`-L diretório`

Especifica o diretório onde o interpretador da linguagem deve ser encontrado. Normalmente o diretório é encontrado automaticamente; esta opção é principalmente para fins de depuração.

O createlang também aceita os seguintes argumentos de linha de comando para os parâmetros de conexão:

`-h hospedeiro`

`--host hospedeiro`

Especifica o nome de hospedeiro da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/) é usado como o diretório do soquete do domínio Unix.

`-p porta`

`--port porta`

Especifica a porta TCP, ou a extensão de arquivo do soquete do domínio Unix local, onde o servidor está atendendo as conexões.

```
-U nome_do_usuario  
--username nome_do_usuario
```

Nome do usuário para conectar.

```
-W  
--password
```

Força a solicitação da senha.

## Ambiente

```
PGDATABASE  
PGHOST  
PGPORT  
PGUSER
```

Parâmetros de conexão padrão.

## Diagnósticos

As mensagens de erro são auto-explicativas, em sua maioria. Caso não seja, execute o `createlang` com a opção `--echo` e consulte o respectivo comando SQL para obter detalhes.

## Observações

Use `droplang` para remover uma linguagem.

## Exemplos

Para instalar a linguagem `pltcl` no banco de dados `template1`:

```
$ createlang pltcl template1
```

## Consulte também

`droplang`, *CREATE LANGUAGE*

# createuser

## Nome

`createuser` — cria uma nova conta de usuário do PostgreSQL

## Sinopse

`createuser` [*opção...*] [*nome\_do\_usuario*]

## Descrição

O utilitário `createuser` cria um novo usuário do PostgreSQL. Somente os superusuários (usuários com `usesuper` definido na tabela `pg_shadow`) podem criar novos usuários do PostgreSQL e, portanto, o `createuser` deve ser executado por alguém que possa se conectar como superusuário do PostgreSQL.

Ser um superusuário também implica na habilidade de transpor as verificações de permissão de acesso do banco de dados e, portanto, o privilégio de superusuário deve ser concedido criteriosamente.

O `createuser` é uma capa em torno do comando `CREATE USER` do SQL. Não existe diferença efetiva entre criar usuários através deste utilitário, ou através de outros métodos para acessar o servidor.

## Opções

O `createuser` aceita os seguintes argumentos de linha de comando:

*nome\_do\_usuario*

Especifica o nome do usuário do PostgreSQL a ser criado. O nome deve ser único entre todos os usuários do PostgreSQL.

`-a`

`--adduser`

Permite o novo usuário criar outros usuários (Nota: na verdade isto torna o novo usuário um *superusuário*. Esta opção possui um nome equivocado).

`-A`

`--no-adduser`

O novo usuário não pode criar outros usuários (ou seja, o novo usuário é um usuário normal, e não um superusuário). Este é o padrão.

`-d`

`--createdb`

O novo usuário pode criar bancos de dados.

`-D`

`--no-createdb`

O novo usuário não pode criar bancos de dados. Este é o padrão.

`-e`

`--echo`

Mostra os comandos que o `createuser` gera e envia para o servidor.

`-E`

`--encrypted`

Criptografa a senha do usuário armazenada no banco de dados. Se não for especificado, será utilizado o comportamento padrão para senhas.

`-i número`  
`--sysid número`

Permite escolher uma identificação diferente do padrão para o novo usuário. Não é necessário, mas algumas pessoas gostam.

`-N`  
`--unencrypted`

Não criptografa a senha do usuário armazenada no banco de dados. Se não for especificado, será utilizado o comportamento padrão para senhas.

`-P`  
`--pwprompt`

Se for usado, o createuser solicita a senha do novo usuário. Não é necessário caso não se planeje usar autenticação por senha.

`-q`  
`--quiet`

Não exibe resposta.

Será solicitado o nome e outras informações que estejam faltando, se não forem especificadas na linha de comando.

O createuser também aceita os seguintes argumentos de linha de comando para os parâmetros de conexão:

`-h hospedeiro`  
`--host hospedeiro`

Especifica o nome de hospedeiro da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é usado como o diretório do soquete do domínio Unix.

`-p porta`  
`--port porta`

Especifica a porta TCP, ou a extensão de arquivo do soquete do domínio Unix local, onde o servidor está atendendo as conexões.

`-U nome_do_usuario`  
`--username nome_do_usuario`

Nome do usuário para conectar (e não o nome do usuário a ser criado).

`-W`  
`--password`

Força a solicitação da senha (para conectar ao servidor, e não a senha do novo usuário).

## Ambiente

PGHOST  
 PGPORT  
 PGUSER

Parâmetros de conexão padrão.

## Diagnósticos

Havendo dificuldade, veja no comando *CREATE USER* e no *psql* a discussão dos problemas possíveis e as mensagens de erro. O servidor de banco de dados deve estar executando no hospedeiro de destino. Também se aplicam todas as definições de conexão padrão e as variáveis de ambiente utilizadas pela biblioteca cliente libpq.

## Exemplos

Para criar o usuário `joe1` no servidor de banco de dados padrão:

```
$ createuser joel
Is the new user allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
```

Criar o mesmo usuário joel usando o servidor no hospedeiro eden, porta 5000, evitando solicitação de informação e vendo o comando subjacente:

```
$ createuser -p 5000 -h eden -D -A -e joel
CREATE USER "joel" NOCREATEDB NOCREATEUSER
CREATE USER
```

## Consulte também

dropuser, *CREATE USER*

# dropdb

## Nome

dropdb — remove um banco de dados do PostgreSQL

## Sinopse

dropdb [*opção...*] *nome\_do\_banco\_de\_dados*

## Descrição

O utilitário dropdb remove um banco de dados do PostgreSQL. Para executar este comando é necessário ser um superusuário, ou o dono do banco de dados.

O dropdb é uma capa em torno do comando *DROP DATABASE* do SQL. Não existe diferença efetiva entre remover bancos de dados através deste utilitário, ou através de outros métodos para acessar o servidor.

## Opções

O dropdb aceita os seguintes argumentos de linha de comando:

*nome\_do\_banco\_de\_dados*

Especifica o nome do banco de dados a ser removido.

-e

--echo

Mostra os comandos que o dropdb gera e envia para o servidor.

-i

--interactive

Solicita a confirmação antes de fazer qualquer operação destrutiva.

-q

--quiet

Não exibe resposta.

O dropdb também aceita os seguintes argumentos de linha de comando para os parâmetros de conexão:

-h *hospedeiro*

--host *hospedeiro*

Especifica o nome de hospedeiro da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é usado como o diretório do soquete do domínio Unix.

-p *porta*

--port *porta*

Especifica a porta TCP, ou a extensão de arquivo do soquete do domínio Unix local, onde o servidor está atendendo as conexões.

-U *nome\_do\_usuario*

--username *nome\_do\_usuario*

Nome do usuário para conectar.

-W

--password

Força a solicitação da senha.

## Ambiente

PGHOST  
PGPORT  
PGUSER

Parâmetros de conexão padrão.

## Diagnósticos

Havendo dificuldade, veja no comando *DROP DATABASE* e no *psql* a discussão dos problemas possíveis e as mensagens de erro. O servidor de banco de dados deve estar executando no hospedeiro de destino. Também se aplicam todas as definições de conexão padrão e as variáveis de ambiente utilizadas pela biblioteca cliente *libpq*.

## Exemplos

Para remover o banco de dados `demo` no servidor de banco de dados padrão:

```
$ dropdb demo
DROP DATABASE
```

Para remover o banco de dados `demo` usando o servidor no hospedeiro `eden`, porta 5000, com confirmação e vendo o comando subjacente:

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE "demo"
DROP DATABASE
```

## Consulte também

`createdb`, *DROP DATABASE*

# droplang

## Nome

droplang — remove uma linguagem procedural do PostgreSQL

## Sinopse

```
droplang [opção_de_conexão...] nome_da_linguagem [nome_do_banco_de_dados]
droplang [opção_de_conexão...] --list | -l nome_do_banco_de_dados
```

## Descrição

O droplang é um utilitário para remover, de um banco de dados do PostgreSQL, uma linguagem de programação existente. O droplang pode remover qualquer linguagem procedural, até mesmo as não fornecidas na distribuição do PostgreSQL.

Embora as linguagens de programação do servidor possam ser removidas diretamente usando vários comandos SQL, recomenda-se usar o aplicativo droplang, porque este realiza várias verificações, e é muito mais fácil usá-lo. Consulte o comando *DROP LANGUAGE* para obter informações adicionais.

## Opções

O droplang aceita os seguintes argumentos de linha de comando:

*nome\_da\_linguagem*

Especifica o nome da linguagem de programação do servidor a ser removida.

`[-d] nome_do_banco_de_dados`

`--dbname] nome_do_banco_de_dados`

Especifica de qual banco de dados a linguagem deve ser removida. O padrão é usar o banco de dados com o mesmo nome do usuário corrente do sistema operacional.

`-e`

`--echo`

Mostra os comandos SQL à medida que são executados.

`-l`

`--list`

Exibe a lista de linguagens instaladas no banco de dados de destino.

O droplang também aceita os seguintes argumentos de linha de comando para os parâmetros de conexão:

`-h hospedeiro`

`--host hospedeiro`

Especifica o nome de hospedeiro da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é usado como o diretório do soquete do domínio Unix.

`-p porta`

`--port porta`

Especifica a porta TCP, ou a extensão de arquivo do soquete do domínio Unix local, onde o servidor está atendendo as conexões.

`-U nome_do_usuario`

`--username nome_do_usuario`

Nome do usuário para conectar.



-W  
--password

Força a solicitação da senha.

## Ambiente

PGDATABASE  
PGHOST  
PGPORT  
PGUSER

Parâmetros de conexão padrão.

## Diagnósticos

As mensagens de erro são, em sua maioria, auto-explicativas. Se alguma não for, execute o droplang com a opção `--echo` e consulte o respectivo comando SQL para obter detalhes.

## Observações

Use createlang para adicionar uma linguagem.

## Exemplos

Para remover a linguagem `pltcl`:

```
$ droplang pltcl template1
```

## Consulte também

createlang, *DROP LANGUAGE*

# dropuser

## Nome

dropuser — remove uma conta de usuário do PostgreSQL

## Sinopse

dropuser [*opção...*] [*nome\_do\_usuario*]

## Descrição

O utilitário dropuser remove um usuário do PostgreSQL, e os bancos de dados que este usuário possui. Somente os superusuários (usuários com `usesuper` definido na tabela `pg_shadow`) podem remover usuários do PostgreSQL.

O dropuser é uma capa em torno do comando `DROP USER` do SQL. Não existe diferença efetiva entre remover usuários através deste utilitário, ou através de outros métodos para acessar o servidor.

## Opções

O dropuser aceita os seguintes argumentos de linha de comando:

*nome\_do\_usuario*

Especifica o nome do usuário do PostgreSQL a ser removido. Será solicitado o nome caso não seja especificado na linha de comando.

`-e`

`--echo`

Mostra os comandos que o dropuser gera e envia para o servidor.

`-i`

`--interactive`

Solicita a confirmação antes de remover o usuário.

`-q`

`--quiet`

Não exibe resposta.

O dropuser também aceita os seguintes argumentos de linha de comando para os parâmetros de conexão:

`-h hospedeiro`

`--host hospedeiro`

Especifica o nome de hospedeiro da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/), é usado como o diretório do soquete do domínio Unix.

`-p porta`

`--port porta`

Especifica a porta TCP, ou a extensão de arquivo do soquete do domínio Unix local, onde o servidor está atendendo as conexões.

`-U nome_do_usuario`

`--username nome_do_usuario`

Nome do usuário para conectar (e não o nome do usuário a ser removido).

`-W`

`--password`

Força a solicitação da senha (para conectar ao servidor, e não a senha do usuário a ser removido).

## Ambiente

PGHOST  
PGPORT  
PGUSER

Parâmetros de conexão padrão.

## Diagnósticos

Havendo dificuldade, veja no comando *DROP USER* e no psql a discussão dos problemas possíveis e as mensagens de erro. O servidor de banco de dados deve estar executando no hospedeiro de destino. Também se aplicam todas as definições de conexão padrão e as variáveis de ambiente utilizadas pela biblioteca cliente libpq.

## Exemplos

Para remover o usuário joel do servidor de banco de dados padrão:

```
$ dropuser joel
DROP USER
```

Para remover o usuário joel usando o servidor no hospedeiro eden, porta 5000, com confirmação e vendo o comando subjacente:

```
$ dropuser -p 5000 -h eden -i -e joel
User "joel" and any owned databases will be permanently deleted.
Are you sure? (y/n) y
DROP USER "joel"
DROP USER
```

## Consulte também

createuser, *DROP USER*

# ecpg

## Nome

ecpg — pré-processador da linguagem SQL incorporada para a linguagem C

## Sinopse

ecpg [*opção...*] *arquivo...*

## Descrição

O utilitário `ecpg` é o pré-processador da linguagem SQL incorporada (*embedded*) para programas escritos na linguagem C. Converte programas C com declarações SQL incorporadas em código C normal, substituindo as chamadas ao SQL por chamadas a funções especiais. Os arquivos de saída podem, então, ser processados por qualquer cadeia de ferramentas do compilador C.

O `ecpg` converte cada arquivo de entrada especificado na linha de comando, no arquivo de saída C correspondente. De preferência, os arquivos de entrada devem possuir a extensão `.pgc` e, neste caso, a extensão é substituída por `.c` para determinar o nome do arquivo de saída. Se a extensão do arquivo de entrada não for `.pgc`, então o nome do arquivo de saída será gerado anexando `.c` ao nome completo do arquivo de entrada. O nome do arquivo de saída pode ser especificado por meio da opção `-o`.

Esta página de referência não descreve a linguagem SQL incorporada. Consulte o Capítulo 29 para obter informações adicionais sobre este tópico.

## Opções

O `ecpg` aceita os seguintes argumentos de linha de comando:

`-c`

Gera, automaticamente, certos códigos C a partir do código SQL. Atualmente funciona para `EXEC SQL TYPE`.

`-C modo`

Define o modo de compatibilidade. O *modo* pode ser `INFORMIX` ou `INFORMIX_SE`.

`-D símbolo`

Define um símbolo do pré-processador C.

`-i`

Analisa, também, os arquivos de inclusão do sistema.

`-I diretório`

Especifica um caminho de inclusão adicional, utilizado para encontrar arquivos incluídos por meio de `EXEC SQL INCLUDE`. Por padrão os seguintes: `.` (o diretório atual), `/usr/local/include`, o diretório de inclusão do PostgreSQL definido em tempo de compilação (por padrão `/usr/local/pgsql/include`) e `/usr/include`, nesta ordem.

`-o arquivo_de_saída`

Especifica que o `ecpg` deve escrever toda a sua saída no *arquivo\_de\_saída* especificado.

`-r opção`

Seleciona um comportamento em tempo de execução. Atualmente *opção* pode ser apenas `no_indicator`.

`-t`

Ativa a auto-efetivação (`auto-commit`) das transações. Neste modo, cada comando SQL é automaticamente efetivado, a não ser que esteja dentro de um bloco de transação explícito. No modo padrão, os comandos são efetivados somente quando é emitido `EXEC SQL COMMIT`.

`-v`

Mostra informações adicionais, incluindo a versão e o caminho de inclusão.

`--help`

Mostra um breve resumo da utilização e depois termina.

`--version`

Mostra a informação da versão e depois termina.

## Observações

Ao compilar arquivos com o código C pré-processado, o compilador necessita encontrar os arquivos de cabeçalho do ECPG no diretório de inclusão do PostgreSQL. Portanto, é necessário usar a opção `-I` ao chamar o compilador (por exemplo, `-I/usr/local/pgsql/include`).

Os programas escritos em C com comandos SQL incorporados necessitam da biblioteca `libecpg` para a ligação. Pode ser usado, por exemplo, as opções do ligador `-L/usr/local/pgsql/lib -lecpg`.

Os nomes destes diretórios, apropriados para a instalação, podem ser descobertos utilizando o aplicativo `pg_config`.

## Exemplos

Havendo um arquivo fonte C chamado `prog1.pgc`, com comandos SQL incorporados, pode ser criado um programa executável utilizando a seguinte seqüência de comandos:

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

# pg\_config

## Nome

`pg_config` — retorna informações sobre a versão do PostgreSQL instalada

## Sinopse

```
pg_config {--bindir | --includedir | --includedir-server | --libdir | --pkglibdir | --pgxs | --configure | --version...}
```

## Descrição

O utilitário `pg_config` mostra os parâmetros de configuração da versão do PostgreSQL atualmente instalada. Sua finalidade é, por exemplo, ser usado por pacotes de software que querem interfacear com o PostgreSQL, para ajudar a encontrar os arquivos de cabeçalho e bibliotecas necessários.

## Opções

Para usar o `pg_config` deve-se fornecer uma ou mais das seguintes opções:

`--bindir`

Mostra o local onde se encontram os executáveis do usuário. É usado, por exemplo, para encontrar o aplicativo `psql`. Normalmente, este é também o local onde o `pg_config` reside.

`--includedir`

Mostra o local onde se encontram os arquivos de cabeçalho da linguagem C das interfaces cliente.

`--includedir-server`

Mostra ao local onde se encontram os arquivos de cabeçalho da linguagem C para programação do servidor.

`--libdir`

Mostra o local onde se encontram as bibliotecas de código objeto.

`--pkglibdir`

Mostra o local onde se encontram os módulos carregáveis dinamicamente, ou onde o servidor deve procurá-los (Também podem estar instalados neste diretório outros arquivos de dados dependentes da arquitetura).

`--pgxs`

Mostra o local onde se encontram os arquivos `Makefile` das extensões.

`--configure`

Mostra as opções passadas para o script `configure` quando o PostgreSQL foi configurado para ser gerado. Pode ser utilizado para reproduzir uma configuração idêntica, ou para descobrir com quais opções o pacote binário foi construído; entretanto, deve ser observado que os pacotes binários geralmente contêm correções específicas da distribuição.

`--version`

Mostra a versão do PostgreSQL e termina.

Se for fornecida mais de uma opção (exceto `--version`), a informação é mostrada nesta ordem, um item por linha.

## Observações

A opção `--includedir-server` começou no PostgreSQL 7.2. Nas versões anteriores, os arquivos de inclusão do servidor estavam instalados no mesmo local dos cabeçalhos dos clientes, que podia ser consultado pela opção `--includedir`. Para tratar os dois casos, deve-se tentar primeiro a opção mais nova, e testar o status da saída para verificar se a execução foi bem-sucedida.

Nas versões do PostgreSQL anteriores a 7.1, antes do comando `pg_config` existir, não existia um método equivalente para encontrar as informações de configuração.

## **Histórico**

O `pg_config` apareceu pela primeira vez no PostgreSQL 7.1.

# pg\_dump

## Nome

`pg_dump` — salva um banco de dados do PostgreSQL em um arquivo de script ou de outro tipo

## Sinopse

`pg_dump [opção...] [nome_do_banco_de_dados]`

## Descrição

O `pg_dump` é um utilitário para fazer cópia de segurança de um banco de dados do PostgreSQL. São feitas cópias de segurança consistentes mesmo que o banco de dados esteja sendo utilizado concorrentemente. O `pg_dump` não bloqueia os outros usuários que estão acessando o banco de dados (leitura ou escrita).

As cópias de segurança podem ser feitas no formato de script ou em outros formatos. As cópias de segurança no formato de script são arquivos no formato texto puro, contendo os comandos SQL necessários para reconstruir o banco de dados no estado em que este se encontrava quando foi salvo. Para restaurar a partir destes scripts, deve ser utilizado o `psql`. Os arquivos de script podem ser utilizados para reconstruir o banco de dados até em outras máquinas com outras arquiteturas; com algumas modificações, até mesmo em outros produtos gerenciadores de banco de dados SQL.

Os formatos de arquivo de cópia de segurança alternativos devem ser utilizados com o `pg_restore` para reconstruir o banco de dados. Estes formatos permitem que o `pg_restore` selecione o que será restaurado, ou mesmo reordene os itens antes de restaurá-los. Os formatos alternativos de cópia de segurança também permitem salvar e restaurar os “objetos grandes”, o que não é possível com a cópia de segurança em arquivo de script. As formatos de cópia de segurança alternativos também são projetados para serem portáteis entre arquiteturas diferentes.

Quando usado com um dos formatos de cópia de segurança alternativos, e combinado com o `pg_restore`, o `pg_dump` fornece um mecanismo flexível para cópias de segurança e transferência. O `pg_dump` pode ser usado para fazer a cópia de segurança de todo o banco de dados e, posteriormente, o `pg_restore` pode ser usado para examinar a cópia de segurança e/ou selecionar as partes do banco de dados a serem restauradas. O formato de arquivo de saída mais flexível é o “personalizado” (`custom`, `-Fc`); permite a seleção e a reordenação de todos os itens da cópia de segurança, e é comprimido por padrão. O formato `tar` (`-Ft`) não é comprimido e não permite reordenar os dados ao a restaurar, mas por outro lado é bastante flexível; além disso, pode ser manipulado pelas ferramentas padrão do Unix, como o `tar`.

Ao executar o `pg_dump` a saída deve ser examinada à procura de advertências (escritas na saída de erro padrão), com atenção especial às limitações mostradas abaixo.

## Opções

As seguintes opções de linha de comando controlam o conteúdo e o formato da saída:

`nome_do_banco_de_dados`

Especifica o nome do banco de dados a ser salvo. Se não for especificado, é utilizada a variável de ambiente `PGDATABASE`. Caso esta variável não esteja definida, é utilizado o nome do usuário especificado para a conexão.

`-a`

`--data-only`

Salva somente os dados, não salva o esquema (definições de dado).

Esta opção só faz sentido para o formato texto-puro. Para os formatos alternativos esta opção pode ser especificada ao chamar o `pg_restore`.

`-b`

`--blobs`

Inclui os objetos grandes na cópia de segurança. Deve ser selecionado um formato de saída não-texto.



-c

--clean

Inclui comandos para remover (`drop`) os objetos do banco de dados antes dos comandos para criá-los.

Esta opção só faz sentido para o formato texto-puro. Para os formatos alternativos esta opção pode ser especificada ao chamar o `pg_restore`.

-C

--create

Inicia a saída por um comando para criar o banco de dados e conectar ao banco de dados criado (Com um script assim não importa qual banco de dados se está conectado antes de executar o script).

Esta opção só faz sentido para o formato texto-puro. Para os formatos alternativos a opção pode ser especificada ao chamar o `pg_restore`.

-d

--inserts

Salva os dados como comandos `INSERT`, em vez de `COPY`. Torna a restauração muito lenta; sua utilização principal é para fazer cópias de segurança que possam ser carregadas em outros bancos de dados que não o PostgreSQL. Deve ser observado que a restauração pode falhar inteiramente se a ordem das colunas tiver sido modificada. A opção `-D` é mais segura, mas ainda mais lenta.

-D

--column-inserts

--attribute-inserts

Salva os dados como comandos `INSERT` explicitando os nomes das colunas (`INSERT INTO tabela (coluna, ...) VALUES ...`). Torna a restauração muito lenta; sua utilização principal é para fazer cópias de segurança que possam ser carregadas em outros bancos de dados que não o PostgreSQL.

-f *arquivo*--file=*arquivo*

Envia a saída para o arquivo especificado. Se for omitido é usada a saída padrão.

-F *formato*--format=*formato*

Seleciona o formato da saída. O *formato* pode ser um dos seguintes:

p

Gera um arquivo de script SQL no formato texto-puro (padrão)

t

Gera um arquivo `tar` adequado para servir de entrada para o `pg_restore`. A utilização deste formato de arquivo permite reordenar e/ou excluir objetos do banco de dados ao fazer a restauração. Também é possível limitar os dados a serem recarregados ao fazer a restauração.

c

Gera um arquivo personalizado adequado para servir de entrada para o `pg_restore`. Este é o formato mais flexível, porque permite a reordenação da restauração dos dados, assim como das definições dos objetos. Também, este formato é comprimido por padrão.

-i

--ignore-version

Ignora a diferença de versão entre o `pg_dump` e o servidor de banco de dados.

O `pg_dump` pode tratar bancos de dados de versões anteriores do PostgreSQL, mas as versões muito antigas não são mais suportadas (atualmente as anteriores a 7.0). Esta opção deve ser utilizada se for necessário desconsiderar a verificação de versão (mas se o `pg_dump` não for bem-sucedido, não diga que não foi avisado).

`-n esquema`

`--schema=esquema`

Salva apenas o conteúdo do *esquema*. Se esta opção não for especificada, todos os esquemas no banco de dados especificado (fora os do sistema) são salvos.

**Nota:** Neste modo, o `pg_dump` não tenta salvar os demais objetos de banco de dados que os objetos no esquema selecionado possam depender. Portanto, não existe nenhuma garantia que o resultado de salvar um único esquema possa, por si próprio, ser bem-sucedido quando restaurado em um banco de dados vazio.

`-o`

`--oids`

Salva os identificadores de objeto (OIDs) de todas as tabelas como parte dos dados. Esta opção deve ser usada quando a coluna OID é referenciada de alguma maneira (por exemplo, em uma restrição de chave estrangeira). Caso contrário, esta opção não deve ser usada.

`-O`

`--no-owner`

Não gera comandos para definir o dono dos objetos correspondendo ao do banco de dados original. Por padrão, o `pg_dump` emite os comandos `ALTER OWNER` ou `SET SESSION AUTHORIZATION` para definir o dono dos objetos de bancos de dados criados. Estes comandos não são bem-sucedidos quando o script é executado, a menos que o script seja executado por um superusuário (ou o mesmo usuário que possui todos os objetos presentes no script). Para gerar um script que possa ser restaurado por qualquer usuário, mas que torna este usuário o dono de todos os objetos, deve ser especificada a opção `-O`.

Esta opção só faz sentido para o formato texto-puro. Para os formatos alternativos a opção pode ser especificada ao chamar o `pg_restore`.

`-R`

`--no-reconnect`

Esta opção está obsoleta, mas ainda é aceita para manter compatibilidade com as versões anteriores.

`-s`

`--schema-only`

Salva somente o esquema (definições dos dados), não os dados.

`-S nome_do_usuario`

`--superuser=nome_do_usuario`

Especifica o nome de usuário do superusuário a ser usado para desabilitar os gatilhos. Somente é relevante quando é usada a opção `--disable-triggers` (Geralmente é melhor não utilizar esta opção e, em vez disso, executar o script produzido como um superusuário).

`-t tabela`

`--table=tabela`

Salva somente os dados da *tabela*. É possível existirem várias tabelas com o mesmo nome em esquemas diferentes; se este for o caso, todas as tabelas correspondentes serão salvas. Deve ser especificado tanto `--schema` quanto `--table` para selecionar apenas uma tabela.

**Nota:** Neste modo, o `pg_dump` não tenta salvar os demais objetos de banco de dados que a tabela selecionada possa depender. Portanto, não existe nenhuma garantia que o resultado de salvar uma única tabela possa, por si próprio, ser bem-sucedido quando restaurado em um banco de dados vazio.

`-v`

`--verbose`

Especifica o modo verboso, fazendo o `pg_dump` colocar comentários detalhados sobre os objetos e os tempos de início/fim no arquivo de cópia de segurança, e mensagens de progresso na saída de erro padrão.

```
-x
--no-privileges
--no-acl
```

Impede salvar os privilégios de acessos (comandos GRANT/REVOKE).

```
-X disable-dollar-quoting
--disable-dollar-quoting
```

Esta opção desabilita a utilização do caractere cifrão (\$) para delimitar o corpo das funções, obrigando a utilização da sintaxe para cadeia de caracteres do padrão SQL.

```
-X disable-triggers
--disable-triggers
```

Esta opção somente é relevante ao criar um arquivo de cópia de segurança somente de dados. Faz o pg\_dump incluir comandos para desabilitar, temporariamente, os gatilhos das tabelas de destino enquanto os dados são recarregados. Deve ser utilizado quando existem verificações de integridade referencial, ou outros gatilhos nas tabelas, que não se deseja que sejam chamados durante a recarga dos dados.

Atualmente, os comandos emitidos para a opção --disable-triggers devem ser executados por superusuários. Portanto, também deve ser especificado o nome de um superusuário com a opção -S ou, de preferência, executar, com cuidado, o script produzido como um superusuário.

Esta opção só faz sentido para o formato texto-puro. Para os formatos alternativos esta opção pode ser especificada ao chamar o pg\_restore.

```
-X use-set-session-authorization
--use-set-session-authorization
```

Gera comandos SET SESSION AUTHORIZATION do padrão SQL em vez dos comandos OWNER TO. Isto torna a cópia de segurança mais compatível com o padrão, mas dependendo da disposição dos objetos na cópia de segurança pode não restaurar de forma apropriada.

```
-Z 0..9
--compress=0..9
```

Especifica o nível de compressão a ser usado nas cópias de segurança com formatos que suportam compressão (atualmente somente o formato personalizado suporta compressão).

As seguintes opções de linha de comando controlam os parâmetros de conexão com o servidor de banco de dados:

```
-h hospedeiro
--host=hospedeiro
```

Especifica o nome de hospedeiro da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/) é usado como o diretório do soquete do domínio Unix. O padrão é obter o nome a partir da variável de ambiente PGHOST, se esta estiver definida, senão tentar uma conexão pelo soquete do domínio Unix.

```
-p porta
--port=porta
```

Especifica a porta TCP, ou a extensão do arquivo de soquete do domínio Unix local, onde o servidor está atendendo as conexões. O padrão é obter o valor a partir da variável de ambiente PGPORT, se esta estiver definida, senão usar o valor padrão compilado.

```
-U nome_do_usuario
```

Conectar como o usuário especificado.

```
-W
```

Força a solicitação da senha, o que deve acontecer automaticamente quando o servidor requer autenticação por senha.

## Ambiente

```
PGDATABASE
PGHOST
PGPORT
PGUSER
```

Parâmetros de conexão padrão.

## Diagnósticos

O `pg_dump` executa internamente comandos `SELECT`. Se acontecerem problemas ao executar o `pg_dump`, deve-se ter certeza que é possível selecionar as informações no banco de dados utilizando, por exemplo, o utilitário `psql`.

## Observações

Se o agrupamento de bancos de dados tiver alguma adição local ao banco de dados `template1`, deve-se ter o cuidado de restaurar a saída do `pg_dump` em um banco de dados totalmente vazio; senão, podem acontecer erros devido à duplicidade de definição dos objetos adicionados. Para criar um banco de dados vazio, sem nenhuma adição local, deve-se fazê-lo partir de `template0`, e não de `template1` como, por exemplo:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

O `pg_dump` possui algumas poucas limitações:

- Ao salvar uma única tabela, ou no formato texto-puro, o `pg_dump` não trata os objetos grandes. Os objetos grandes devem ser salvos juntamente com todo o banco de dados usando um dos formatos de cópia de segurança não-texto.
- Quando é escolhido salvar apenas os dados, e se utiliza a opção `--disable-triggers`, o `pg_dump` emite comandos para desabilitar os gatilhos das tabelas do usuário antes de inserir os dados, e comandos para reabilitá-los após os dados terem sido inseridos. Se a restauração for interrompida antes do fim, os catálogos do sistema podem ser deixados em um estado errado.

Os membros de arquivos `tar` estão limitados a um tamanho inferior a 8 GB (esta limitação é inerente ao formato dos arquivos `tar`). Portanto, este formato não pode ser utilizado se a representação textual de uma tabela exceder este tamanho. O tamanho total do arquivo `tar`, e dos outros formatos de saída, não possui limitação exceto, talvez, pelo sistema operacional.

Os arquivos de cópia de segurança produzidos pelo `pg_dump` não contêm as estatísticas utilizadas pelo otimizador para fazer as decisões de planejamento dos comandos. Portanto, é aconselhável executar o `ANALYZE` após restaurar de uma cópia de segurança para garantir um bom desempenho.

## Exemplos

Para salvar um banco de dados:

```
$ pg_dump meu_bd > db.out
```

Para recarregar este banco de dados:

```
$ psql -d banco_de_dados -f db.out
```

Para salvar o banco de dados chamado `meu_bd` contendo objetos grandes em um arquivo `tar`:

```
$ pg_dump -Ft -b meu_bd > db.tar
```

Para recarregar este banco de dados (com os objetos grandes) em um banco de dados existente chamado `novo_bd`:

```
$ pg_restore -d novo_bd db.tar
```

## Histórico

O `pg_dump` apareceu pela primeira vez no Postgres95 versão 0.02. Os formatos de saída não-texto-puro foram introduzidos no PostgreSQL versão 7.1.

## Consulte também

`pg_dumpall`, `pg_restore`, `psql`

# pg\_dumpall

## Nome

`pg_dumpall` — salva os bancos de dados de um agrupamento do PostgreSQL em um arquivo de script

## Sinopse

`pg_dumpall` [*opção...*]

## Descrição

O `pg_dumpall` é um utilitário para salvar (`dump`) todos os bancos de dados de um agrupamento do PostgreSQL em um arquivo de script. O arquivo de script contém comandos SQL que podem ser usados como entrada do `psql` para restaurar os bancos de dados. Isto é feito chamando o `pg_dump` para cada banco de dados do agrupamento. O `pg_dumpall` também salva os objetos globais, comuns a todos os bancos de dados (O `pg_dump` não salva estes objetos). Atualmente são incluídas informações sobre os usuários do banco de dados e grupos, e permissões de acesso aplicadas aos bancos de dados como um todo.

Portanto, o `pg_dumpall` é uma solução integrada para realizar cópias de segurança dos bancos de dados. Entretanto, deve ser observada a seguinte limitação: não é possível salvar “objetos grandes”, porque o `pg_dump` não pode salvar estes objetos em arquivos texto. Havendo bancos de dados contendo objetos grandes, estes devem ser salvos usando um dos modos de saída não-texto do `pg_dump`.

Como o `pg_dumpall` lê tabelas de todos os bancos de dados, muito provavelmente será necessário se conectar como um superusuário para poder gerar uma cópia completa. Também será necessário o privilégio de superusuário para executar o script produzido, para poder criar usuários e grupos, e para poder criar os bancos de dados.

O script SQL é escrito na saída padrão. Devem ser usados operadores de linha de comando para redirecionar para um arquivo.

O `pg_dumpall` precisa se conectar várias vezes ao servidor PostgreSQL (uma vez para cada banco de dados). Se for utilizada autenticação por senha, provavelmente será solicitada a senha cada uma destas vezes. Neste caso é conveniente existir o arquivo `$HOME/.pgpass`. Consulte a Seção 27.12 para obter informações adicionais.

## Opções

As seguintes opções de linha de comando controlam o conteúdo e o formato da saída:

`-a`

`--data-only`

Salva somente os dados, não salva o esquema (definições de dado).

`-c`

`--clean`

Inclui comandos para remover (`drop`) os objetos do banco de dados antes dos comandos para criá-los.

`-d`

`--inserts`

Salva os dados como comandos `INSERT`, em vez de `COPY`. Torna a restauração muito lenta; sua utilização principal é para fazer cópias de segurança que possam ser carregadas em outros bancos de dados que não o PostgreSQL. Deve ser observado que a restauração pode falhar inteiramente se a ordem das colunas tiver sido modificada. A opção `-D` é mais segura, mas ainda mais lenta.

```
-D
--column-inserts
--attribute-inserts
```

Salva os dados como comandos INSERT explicitando os nomes das colunas (INSERT INTO *tabela* (*coluna*, ...) VALUES ...). Torna a restauração muito lenta; sua utilização principal é para fazer cópias de segurança que possam ser carregadas em outros bancos de dados que não o PostgreSQL.

```
-g
--globals-only
```

Salva somente os objetos globais (usuários e grupos), e não os banco de dados.

```
-i
--ignore-version
```

Ignora a diferença de versão entre o pg\_dump e o servidor de banco de dados.

O pg\_dump pode tratar bancos de dados de versões anteriores do PostgreSQL, mas as versões muito antigas não são mais suportadas (atualmente as anteriores a 7.0). Esta opção deve ser utilizada se for necessário desconsiderar a verificação de versão (mas se o pg\_dumpall não for bem-sucedido, não diga que não foi avisado).

```
-o
--oids
```

Salva os identificadores de objeto (OIDs) de todas as tabelas como parte dos dados. Esta opção deve ser usada quando a coluna OID é referenciada de alguma maneira (por exemplo, em uma restrição de chave estrangeira). Caso contrário, esta opção não deve ser usada.

```
-O
--no-owner
```

Não gera comandos para definir o dono dos objetos correspondendo ao do banco de dados original. Por padrão, o pg\_dump emite os comandos ALTER OWNER ou SET SESSION AUTHORIZATION para definir o dono dos objetos de bancos de dados criados. Estes comandos não são bem-sucedidos quando o script é executado, a menos que o script seja executado por um superusuário (ou o mesmo usuário que possui todos os objetos presentes no script). Para gerar um script que possa ser restaurado por qualquer usuário, mas que torna este usuário o dono de todos os objetos, deve ser especificada a opção -O.

```
-s
--schema-only
```

Salva somente o esquema (definições dos dados), não os dados.

```
-S nome_do_usuario
--superuser=nome_do_usuario
```

Especifica o nome de usuário do superusuário a ser usado para desabilitar os gatilhos. Somente é relevante quando é usado --disable-triggers (Geralmente é melhor não utilizar esta opção e, em vez disso, executar o script produzido como um superusuário).

```
-v
--verbose
```

Especifica o modo verboso, fazendo o pg\_dump colocar os tempos de início/fim no arquivo de cópia de segurança, e mensagens de progresso na saída de erro padrão. Também habilita a saída verbosa no pg\_dump.

```
-x
--no-privileges
--no-acl
```

Impede salvar os privilégios de acessos (comandos GRANT/REVOKE).

```
-X disable-dollar-quoting
--disable-dollar-quoting
```

Esta opção desabilita a utilização do caractere cifrão (\$) para delimitar o corpo das funções, obrigando a utilização da sintaxe para cadeia de caracteres do padrão SQL.

```
-X disable-triggers
--disable-triggers
```

Esta opção somente é relevante ao criar um arquivo de cópia de segurança somente de dados. Faz o pg\_dumpall incluir comandos para desabilitar, temporariamente, os gatilhos das tabelas de destino enquanto os dados são recarregados. Deve ser utilizado quando existem verificações de integridade referencial, ou outros gatilhos nas tabelas, que não se deseja que sejam chamados durante a recarga dos dados.

Atualmente, os comandos emitidos para a opção `--disable-triggers` devem ser executados por superusuários. Portanto, também deve ser especificado o nome de um superusuário com a opção `-S` ou, de preferência, executar, com cuidado, o script produzido como um superusuário.

```
-X use-set-session-authorization
--use-set-session-authorization
```

Gera comandos SET SESSION AUTHORIZATION do padrão SQL em vez dos comandos OWNER TO. Isto torna a cópia de segurança mais compatível com o padrão, mas dependendo da disposição dos objetos na cópia de segurança pode não restaurar de forma apropriada.

As seguintes opções de linha de comando controlam os parâmetros de conexão com o servidor de banco de dados:

```
-h hospedeiro
```

Especifica o nome de hospedeiro da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/) é usado como o diretório do soquete do domínio Unix. O padrão é obter o nome a partir da variável de ambiente PGHOST, se esta estiver definida, senão tentar uma conexão pelo soquete do domínio Unix.

```
-p porta
```

Especifica a porta TCP, ou a extensão de arquivo do soquete do domínio Unix local, onde o servidor está atendendo as conexões. O padrão é obter o valor a partir da variável de ambiente PGPORT, se esta estiver definida, senão usar o valor padrão compilado.

```
-U nome_do_usuario
```

Conectar como o usuário especificado.

```
-W
```

Força a solicitação da senha, o que deve acontecer automaticamente quando o servidor requer autenticação por senha.

## Ambiente

```
PGHOST
PGPORT
PGUSER
```

Parâmetros de conexão padrão.

## Observações

Como o pg\_dumpall chama o pg\_dump internamente, algumas mensagens de diagnósticos se referem ao pg\_dump.

Ao término da restauração, é aconselhável executar o comando ANALYZE em todos os bancos de dados para que o otimizador tenha estatísticas úteis. Também pode ser executado vacuumdb -a -z para analisar todos os bancos de dados.

## Exemplos

Para salvar todos bancos de dados:

```
$ pg_dumpall > db.out
```

Para recarregar este banco de dados deve ser utilizado, por exemplo:

```
$ psql -f db.out template1
```

(Neste caso o banco de dados a se conectar não tem importância, porque o arquivo de script criado pelo pg\_dumpall contém os comandos apropriados para criar e conectar aos bancos de dados salvos).



## **Consulte também**

`pg_dump`. Veja aí os detalhes sobre as condições de erro possíveis.

# pg\_restore

## Nome

`pg_restore` — restaura um banco de dados do PostgreSQL a partir de um arquivo criado pelo `pg_dump`

## Sinopse

`pg_restore` [*opção...*] [*nome\_da\_cópia\_de\_segurança*]

## Descrição

O `pg_restore` é um utilitário para restaurar um banco de dados do PostgreSQL, a partir de uma cópia de segurança criada pelo `pg_dump` em um dos formatos não-texto-puro. São executados os comandos necessários para reconstruir o banco de dados, no estado em que este se encontrava na hora em que foi salvo. Os arquivos de cópia de segurança também permitem ao `pg_restore` selecionar o que será restaurado, ou mesmo reordenar os itens antes de serem restaurados. Os arquivos de cópia de segurança são projetados para serem portáteis entre arquiteturas diferentes.

O `pg_restore` pode operar de dois modos: Se o nome do banco de dados for especificado, a cópia de segurança é restaurada diretamente no banco de dados (Os objetos grandes só podem ser restaurados utilizando uma conexão direta com o banco de dados como esta). Senão, é criado um script, no formato texto-puro, contendo os comandos SQL necessários para reconstruir o banco de dados (escrito em um arquivo ou na saída padrão), semelhante aos scripts criados pelo `pg_dump`. Algumas das opções que controlam a criação do script são, portanto, análogas às opções do `pg_dump`.

Obviamente, o `pg_restore` não pode restaurar informações que não estejam presentes no arquivo de cópia de segurança. Por exemplo, se a cópia de segurança for gerada usando a opção “salvar dados como comandos `INSERT`”, o `pg_restore` não poderá restaurar os dados usando comandos `COPY`.

## Opções

O `pg_restore` aceita os seguintes argumentos de linha de comando.

*nome\_da\_cópia\_de\_segurança*

Especifica o local do arquivo de cópia de segurança a ser restaurado. Se não for especificado, é usada a entrada padrão.

`-a`

`--data-only`

Salva somente os dados, não salva o esquema (definições de dado).

`-c`

`--clean`

Remove (`drop`) os objetos do banco de dados antes de criá-los.

`-C`

`--create`

Cria o banco de dados antes de restaurá-lo (Quando esta opção é utilizada, o banco de dados especificado na opção `-d` é usado apenas para executar o comando `CREATE DATABASE` inicial. Todos os dados são restaurados no banco de dados cujo nome aparece na cópia de segurança).

`-d nome_do_banco_de_dados`

`--dbname=nome_do_banco_de_dados`

Conecta ao banco de dados *nome\_do\_banco\_de\_dados* e restaura diretamente neste banco de dados.

`-e`

`--exit-on-error`

Termina se for encontrado um erro ao enviar os comandos SQL para o banco de dados. O padrão é continuar e mostrar um contador de erros ao término da restauração.

```
-f arquivo_de_saída
--file=arquivo_de_saída
```

Especifica o arquivo de saída para o script gerado, ou para conter a listagem quando for utilizada a opção `-l`. Por padrão a saída padrão.

```
-F formato
--format=formato
```

Especifica o formato do arquivo da cópia de segurança. Não é necessário especificar o formato, porque o `pg_restore` determina o formato automaticamente. Se for especificado, pode ser um dos seguintes:

t

A cópia de segurança é um arquivo `tar`. Este formato de cópia de segurança permite reordenar e/ou excluir elementos do esquema ao restaurar o banco de dados. Também permite limitar quais dados são recarregados ao restaurar.

c

A cópia de segurança está no formato personalizado do `pg_dump`. Este é o formato mais flexível, porque permite reordenar a restauração dos dados e dos elementos do esquema. Também, este formato é comprimido por padrão.

```
-i
--ignore-version
```

Ignora a verificação da versão do banco de dados.

```
-I nome_do_índice
--index=nome_do_índice
```

Restaura apenas a definição do índice especificado.

```
-l
--list
```

Lista o conteúdo da cópia de segurança. A saída desta operação pode ser usada com a opção `-L` para restringir e reordenar os itens a serem restaurados.

```
-L arquivo_da_listagem
--use-list=arquivo_da_listagem
```

Restaura apenas os elementos presentes no *arquivo\_da\_listagem*, e na ordem que aparecem neste arquivo. As linhas podem ser movidas e, também, podem virar comentário colocando um `;` no seu início (Veja os exemplos abaixo).

```
-O
--no-owner
```

Não gera comandos para definir os donos dos objetos correspondendo aos donos destes objetos no banco de dados de origem. Por padrão, o `pg_restore` executa o comando `ALTER OWNER` ou `SET SESSION AUTHORIZATION` para definir os donos dos elementos criados no esquema. Estes comando não são bem-sucedidos a menos que a conexão inicial com o banco de dados seja feita por um superusuário (ou o mesmo usuário que possui todos os objetos presentes no script). Usando a opção `-O`, pode ser utilizado qualquer nome de usuário na conexão inicial, e este usuário será o dono de todos objetos criados.

```
-P nome_da_função(tipo_do_argumento [, ...])
--function=nome_da_função(tipo_do_argumento [, ...])
```

Restaura apenas a função especificada. Tome cuidado para escrever o nome da função e os argumentos exatamente como estes aparecem na tabela de conteúdo da cópia de segurança.

```
-R
--no-reconnect
```

Esta opção está obsoleta, mas ainda é aceita para manter a compatibilidade com as versões anteriores.

-s

--schema-only

Restaura somente o esquema (definições de dados), não os dados. Os valores das seqüências são reiniciados.

-S *nome\_de\_usuario*

--superuser=*nome\_de\_usuario*

Especifica o nome de usuário do superusuário a ser usado para desabilitar os gatilhos. Somente é relevante quando é usada a opção --disable-triggers.

-t *tabela*

--table=*tabela*

Restaura apenas a definição e/ou dados da tabela especificada.

-T *gatilho*

--trigger=*gatilho*

Restaura apenas o gatilho especificado.

-v

--verbose

Especifica o modo verboso.

-x

--no-privileges

--no-acl

Impede restaurar os privilégios de acessos (comandos GRANT/REVOKE).

-X use-set-session-authorization

--use-set-session-authorization

Gera comandos SET SESSION AUTHORIZATION do padrão SQL, em vez dos comandos OWNER TO. Isto torna a cópia de segurança mais compatível com o padrão, mas dependendo da disposição dos objetos na cópia de segurança pode não restaurar de forma apropriada.

-X disable-triggers

--disable-triggers

Esta opção é relevante apenas quando se restaura somente os dados. Faz com que o pg\_restore execute comandos para desabilitar, temporariamente, os gatilhos das tabelas de destino enquanto os dados são recarregados. Deve ser utilizado quando existem verificações de integridade referencial, ou outros gatilhos nas tabelas, que não se deseja que sejam chamados durante a recarga dos dados.

Atualmente, os comandos emitidos para a opção --disable-triggers devem ser executados por superusuários. Portanto, também deve ser especificado o nome de um superusuário com a opção -S ou, de preferência, executar, com cuidado, o script produzido como um superusuário.

O pg\_restore também aceita os seguintes argumentos de linha de comando para os parâmetros de conexão:

-h *hospedeiro*

--host=*hospedeiro*

Especifica o nome de hospedeiro da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/) é usado como o diretório do soquete do domínio Unix. O padrão é obter o nome a partir da variável de ambiente PGHOST, se esta estiver definida, senão tentar uma conexão pelo soquete do domínio Unix.

-p *porta*

--port=*porta*

Especifica a porta TCP, ou a extensão de arquivo do soquete do domínio Unix local, onde o servidor está atendendo as conexões. O padrão é obter o valor a partir da variável de ambiente PGPORT, se esta estiver definida, senão usar o valor padrão compilado.

`-U nome_de_usuario`

Conectar como o usuário especificado.

`-W`

Força a solicitação da senha, o que deve acontecer automaticamente quando o servidor requer autenticação por senha.

## Ambiente

PGHOST

PGPORT

PGUSER

Parâmetros de conexão padrão.

## Diagnósticos

Quando a conexão direta com o banco de dados é especificada usando a opção `-d`, o `pg_restore` executa internamente comandos SQL. Se acontecerem problemas ao executar o `pg_restore`, deve-se ter certeza que é possível selecionar informações no banco de dados utilizando, por exemplo, o utilitário `psql`.

## Observações

Se o agrupamento de bancos de dados tiver alguma adição local ao banco de dados `template1`, deve-se ter o cuidado de restaurar a saída do `pg_restore` em um banco de dados totalmente vazio; senão, podem acontecer erros devido à duplicidade de definição dos objetos adicionados. Para criar um banco de dados vazio, sem nenhuma adição local, deve-se fazê-lo partir de `template0`, e não de `template1` como, por exemplo:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

As limitações do `pg_restore` estão descritas abaixo.

- Ao restaurar os dados em uma tabela pré-existente utilizando a opção `--disable-triggers`, o `pg_restore` emite comandos para desabilitar os gatilhos das tabelas do usuário antes de inserir os dados, e comandos para reabilitá-los após os dados terem sido inseridos. Se a restauração for interrompida antes do fim, os catálogos do sistema podem ser deixados em um estado errado.
- O `pg_restore` não restaura objetos grandes para uma única tabela. Se a cópia de segurança contém objetos grandes, então todos os objetos grandes são restaurados.

Consulte também a documentação do `pg_dump` para obter os detalhes de suas limitações.

Uma vez restaurado, é aconselhável executar o comando `ANALYZE` em todas as tabelas restauradas para que o otimizador possua estatísticas úteis.

## Exemplos

Para gerar uma cópia de segurança do banco de dados `meu_bd`, que contém objetos grandes, em um arquivo `tar`:

```
$ pg_dump -Ft -b meu_bd > db.tar
```

Para restaurar este banco de dados (com os objetos grandes) no banco de dados chamado `novo_bd`:

```
$ pg_restore -d novo_bd db.tar
```

Para reordenar os itens do banco de dados, primeiro é necessário criar um arquivo contendo a tabela de conteúdo (índice) da cópia de segurança:

```
$ pg_restore -l copia_de_seguranca.arquivo > copia_de_seguranca.list
```

O arquivo de listagem consiste de um cabeçalho e uma linha para cada item como, por exemplo,

```

;
; Archive created at Fri Jul 28 22:28:36 2000
;
;    dbname: birds
;    TOC Entries: 74
;    Compression: 0
;    Dump Version: 1.4-0
;    Format: CUSTOM
;
;
; Selected TOC Entries:
;
2; 145344 TABLE species postgres
3; 145344 ACL species
4; 145359 TABLE nt_header postgres
5; 145359 ACL nt_header
6; 145402 TABLE species_records postgres
7; 145402 ACL species_records
8; 145416 TABLE ss_old postgres
9; 145416 ACL ss_old
10; 145433 TABLE map_resolutions postgres
11; 145433 ACL map_resolutions
12; 145443 TABLE hs_old postgres
13; 145443 ACL hs_old

```

Ponto-e-vírgula inicia um comentário, e os números no início das linhas referem-se aos identificadores internos da cópia de segurança atribuídos a cada item.

As linhas do arquivo podem ser transformadas em comentário, excluídas e reordenadas. Por exemplo

```

10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres

```

poderia ser usado como entrada do pg\_restore e somente restauraria os itens 10 e 6, nesta ordem.

```
$ pg_restore -L copia_de_seguranca.list copia_de_seguranca.arquivo
```

## Histórico

O utilitário pg\_restore apareceu pela primeira vez no PostgreSQL 7.1.

## Consulte também

pg\_dump, pg\_dumpall, psql

# psql

## Nome

psql — terminal interativo do PostgreSQL

## Sinopse

```
psql [opção...] [nome_do_banco_de_dados [nome_do_usuario]]
```

## Descrição

O psql é um cliente no modo terminal do PostgreSQL. Permite digitar comandos interativamente, submetê-los para o PostgreSQL e ver os resultados. Como alternativa, a entrada pode vir de um arquivo. Além disso, disponibiliza vários meta-comandos e diversas funcionalidades semelhantes às do interpretador comandos (`shell`) para facilitar a criação de scripts e automatizar um grande número de tarefas.

## Opções

`-a`

`--echo-all`

Envia todas as linhas de entrada para a saída padrão à medida que são lidas. É mais útil para o processamento de scripts do que no modo interativo. Equivale a definir a variável `ECHO` como `all`.

`-A`

`--no-align`

Comuta para o modo de saída não alinhado (De outra forma, o modo de saída padrão é o alinhado).

`-c comando`

`--command comando`

Especifica que o psql deve executar a cadeia de caracteres *comando* e, em seguida, terminar. Útil em scripts do interpretador de comandos.

O *comando* deve ser uma cadeia de caracteres que possa ser integralmente analisada pelo servidor (ou seja, não contém funcionalidades específicas do psql), ou ser um único comando de contrabarra. Portanto, não podem ser misturados comandos SQL com meta-comandos do psql. Para misturar, a cadeia de caracteres pode ser enviada para o psql conforme mostrado a seguir: `echo "\x \ select * from foo;" | psql`.

Se a cadeia de caracteres do comando contiver vários comandos SQL, estes serão processados em uma única transação, a menos que existam comandos `BEGIN/COMMIT` explícitos incluídos na cadeia de caracteres para dividi-los em várias transações. Este comportamento é diferente do comportamento que ocorre quando a mesma cadeia de caracteres é introduzida na entrada padrão do psql.

`-d nome_do_banco_de_dados`

`--dbname nome_do_banco_de_dados`

Especifica o nome do banco de dados a se conectar. Equivale a especificar *nome\_do\_banco\_de\_dados* como o primeiro argumento não-opção na linha de comando.

`-e`

`--echo-queries`

Copia todos os comandos SQL enviados para o servidor para a saída padrão também. Equivale a definir a variável `ECHO` como `queries`.

`-E`

`--echo-hidden`

Exibe os verdadeiros comandos gerados por `\d` e por outros comandos de contrabarra. Pode ser usado para estudar as operações internas do psql. Equivale a definir a variável `ECHO_HIDDEN` dentro do psql.

```
-f nome_do_arquivo
--file nome_do_arquivo
```

Usa o arquivo *nome\_do\_arquivo* como origem dos comandos, em vez de ler os comandos interativamente. Após processar o arquivo, o psql termina. Sob muitos aspectos equivale ao comando interno \i.

Se o *nome\_do\_arquivo* for - (hífen), então a entrada padrão é lida.

O uso desta opção é sutilmente diferente de escrever `psql < nome_do_arquivo`. De uma maneira geral, as duas formas fazem o esperado, mas o uso da opção `-f` ativa algumas funcionalidades úteis, como mensagens de erro com o número da linha. Ao se usar esta opção existe, também, uma pequena chance de reduzir a sobrecarga de inicialização. Por outro lado, a utilização do redirecionamento da entrada na linha de comando garante, teoricamente, que será produzida exatamente a mesma saída que seria produzida se tudo fosse entrado à mão.

```
-F separador
--field-separator separador
```

Usa o *separador* como separador de campos. Equivale a `\pset fieldsep` ou ao comando \f.

```
-h nome_do_hospedeiro
--host nome_do_hospedeiro
```

Especifica o nome do hospedeiro da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/) é usado como o diretório do soquete do domínio Unix.

```
-H
--html
```

Ativa a saída tabular HTML. Equivale a `\pset format html` ou ao comando \H.

```
-l
--list
```

Mostra todos os bancos de dados disponíveis, e depois termina. As outras opções, fora as de conexão, são ignoradas. Semelhante ao comando interno \list.

```
-o nome_do_arquivo
--output nome_do_arquivo
```

Coloca a saída de todos os comandos no arquivo *nome\_do\_arquivo*. Equivale ao comando \o.

```
-p porta
--port porta
```

Especifica a porta TCP, ou a extensão de arquivo do soquete do domínio Unix local, onde o servidor está atendendo as conexões. O padrão é obter o valor a partir da variável de ambiente PGPORT, se esta estiver definida, senão usar o valor padrão compilado (normalmente 5432).

```
-P atribuição
--pset atribuição
```

Permite especificar opções de exibição no estilo do `\pset` pela linha de comando. Deve ser observado que aqui o nome e o valor devem estar separados pelo sinal de igual, em vez de espaço. Portanto, para definir o formato de saída como LaTeX deve ser escrito `-P format=latex`.

```
-q
--quiet
```

Especifica que o psql deve trabalhar em silêncio. Por padrão, são exibidas mensagens de boas-vindas e várias outras mensagens informativas. Se esta opção for usada, nada disso acontece. É útil em conjunto com a opção `-c`. Dentro do psql é possível definir a variável QUIET para obter o mesmo efeito.

```
-R separador
--record-separator separador
```

Usa o *separador* como separador de registros. Equivale ao comando `\pset recordsep`.



`-s`  
`--single-step`

Executa no modo passo-único, significando que será solicitada uma confirmação antes de cada comando ser enviado para o servidor, com a opção de cancelar a execução. Usado para depurar scripts.

`-S`  
`--single-line`

Executa no modo linha-única, onde o caractere de nova-linha termina o comando SQL, como o ponto-e-vírgula faz.

**Nota:** Este modo é fornecido para aqueles que insistem em usá-lo, mas sua utilização não é incentivada. Em particular, se forem misturados comandos SQL e meta-comandos na mesma linha, a ordem de execução nem sempre será clara para o usuário inexperiente.

`-t`  
`--tuples-only`

Desabilita a exibição dos nomes das colunas, rodapés com contadores de linhas do resultado, etc. É equivalente ao comando `\t`.

`-T opções_de_tabela`  
`--table-attr opções_de_tabela`

Permite especificar opções a serem colocadas dentro da marca `table` do HTML. Consulte `\pset` para obter detalhes.

`-u`

Força o psql solicitar o nome do usuário e a senha antes de conectar ao banco de dados.

Esta opção está obsoleta, porque é conceitualmente incorreta (Solicitar um nome de usuário não padrão e solicitar uma senha porque o servidor requer são realmente duas coisas diferentes). Incentiva-se o uso das opções `-U` e `-W` em seu lugar.

`-U nome_do_usuario`  
`--username nome_do_usuario`

Conecta ao banco de dados como o usuário `nome_do_usuario` em vez do usuário padrão (É necessário ter permissão para fazê-lo, é claro).

`-v atribuição`  
`--set atribuição`  
`--variable atribuição`

Realiza atribuição de variável, como o comando interno `\set`. Deve ser observado que na linha de comando é necessário separar o nome e o valor, se houver, por um sinal de igual. Para remover a definição de uma variável deve ser omitido o sinal de igual. Para apenas definir uma variável, sem um valor, o sinal de igual é usado mas o valor é omitido. Estas atribuições são feitas durante um estágio bem no princípio da inicialização e, portanto, as variáveis reservadas para finalidades internas podem ser sobrescritas posteriormente.

`-V`  
`--version`

Mostra a versão do psql, e depois termina.

`-W`  
`--password`

Força o psql solicitar a senha antes de conectar ao banco de dados.

O psql deve solicitar, automaticamente, a senha sempre que o servidor requerer autenticação por senha. Como atualmente a detecção de solicitação de senha não é inteiramente confiável, esta opção existe para obrigar a solicitação. Se a senha não for solicitada, e o servidor requerer autenticação por senha, a tentativa de conexão não será bem-sucedida.

Esta opção continuará definida por toda a sessão, mesmo que a conexão com o banco de dados seja mudada pelo meta-comando `\connect`.

`-x``--expanded`

Ativa o modo formatação de tabela estendido. Equivale ao comando `\x`.

`-X,``--no-psqlrc`

Não lê o arquivo de inicialização (nem o arquivo `psqlrc` global do sistema, nem o arquivo `~/.psqlrc` do usuário).

`-?``--help`

Mostra a ajuda sobre os argumentos de linha de comando do `psql`, e depois termina.

## Status de saída

O `psql` retorna para o interpretador de comandos: 0 se terminar normalmente; 1 se ocorrer erro fatal próprio (falta de memória, arquivo não encontrado); 2 se a conexão com o servidor teve problema e a sessão não é interativa; 3 se ocorrer erro no script e a variável `ON_ERROR_STOP` estiver definida.

## Utilização

### Conexão com o banco de dados

O `psql` é um aplicativo cliente do PostgreSQL comum. Para conectar a um banco de dados é necessário saber o nome do banco de dados, o nome do hospedeiro, o número da porta do servidor e o nome do usuário a ser usado para conectar. O `psql` pode ser informado sobre estes parâmetros por meio das opções de linha de comando `-d`, `-h`, `-p` e `-U`, respectivamente. Se for encontrado um argumento que não pertence a nenhuma opção, este será interpretado como o nome do banco de dados (ou o nome do usuário, se o nome do banco de dados já tiver sido fornecido). Nem todas estas opções são requeridas; existem padrões úteis. Se for omitido o nome do hospedeiro, então o `psql` se conecta através do soquete do domínio Unix ao servidor no hospedeiro local. O número padrão para a porta é determinado na compilação. Uma vez que o servidor de banco de dados usa o mesmo padrão, não é necessário especificar a porta na maioria dos casos. O nome de usuário padrão é o nome do usuário do Unix, como também é o nome do banco de dados padrão. Deve ser observado que não é possível se conectar a qualquer banco de dados com qualquer nome de usuário. O administrador de banco de dados deve informar as permissões de acesso concedidas.

Quando os padrões não estão como o desejado, a digitação pode ser reduzida definindo as variáveis de ambiente `PGDATABASE`, `PGHOST`, `PGPORT` e/ou `PGUSER` com os valores apropriados (Para as demais variáveis de ambiente consulte a Seção 27.11). Também é conveniente criar o arquivo `~/.pgpass` para evitar ter que digitar regularmente as senhas. Consulte a Seção 27.12 para obter informações adicionais.

Se a conexão não puder ser estabelecida por algum motivo (por exemplo, privilégios insuficientes, o servidor não está executando no hospedeiro de destino, etc.), o `psql` retorna uma mensagem de erro e termina.

### Entrando com comandos SQL

No modo normal de operação, o `psql` exibe um `prompt` com o nome do banco de dados ao qual está conectado, seguido pela cadeia de caracteres `=>`. Por exemplo: <sup>1</sup>

```
$ psql testdb
```

Bem-vindo ao `psql 8.0.0`, o terminal interativo do PostgreSQL.

```
Digite: \copyright para mostrar a licença da distribuição
        \h para ajuda nos comandos SQL
        \? para ajuda nos comandos de contrabarra internos
        \g ou finalizar com ponto-e-vírgula para executar o comando
        \q para sair
```

```
testdb=>
```

No `prompt` o usuário pode digitar comandos SQL. Normalmente, as linhas de entrada são enviadas para o servidor quando é encontrado o caractere ponto-e-vírgula, que termina o comando. Um fim de linha não termina o comando. Portanto, os comandos podem ser distribuídos por várias linhas para maior clareza. Se o comando for enviado e executado sem erro, o resultado do comando será mostrado na tela.

Sempre que um comando é executado, o `psql` também verifica eventos de notificação assíncronos gerados pelo `LISTEN` e `NOTIFY`.

## Meta-Comandos

Qualquer texto digitado no `psql` começando por uma contrabarra (`\`) (não entre apóstrofes, `'`) é um meta-comando do `psql` processado pelo próprio `psql`. Estes comandos ajudam a tornar o `psql` mais útil para administração e para scripts. Os meta-comandos são geralmente chamados de comandos de barra ou de contrabarra.

O formato de um comando `psql` é a contrabarra, seguida imediatamente por um verbo comando, e depois pelos argumentos. Os argumentos são separados do verbo comando, e entre si, por qualquer número de caracteres de espaço em branco.

Para incluir espaço em branco no argumento deve-se colocá-los entre apóstrofes (`'`). Para incluir um apóstrofo neste tipo de argumento, deve-se precedê-lo por uma contrabarra. Qualquer texto entre apóstrofes está sujeito às substituições no estilo C para `\n` (nova-linha), `\t` (tabulação), `\dígito`, `\0dígito` e `\0xdígito` (o caractere com o código decimal, octal ou hexadecimal especificado).

Se um argumento (não entre apóstrofes) começar por dois-pontos (`:`), será considerado como sendo uma variável do `psql`, e o valor desta variável será usado como o argumento.

Os argumentos entre crases (```) são considerados como sendo linhas de comando a serem passadas para o interpretador de comandos. A saída do comando (com o caractere de nova-linha final removido) é usada como o valor do argumento. As seqüências de escape (`\`) acima também se aplicam às crases.

Alguns comandos recebem como argumento um identificador SQL como, por exemplo, o nome de uma tabela. Estes argumentos seguem as regras de sintaxe do SQL: as letras que não se encontram entre aspas (`"`) são transformadas em minúsculas, enquanto as letras que se encontram entre aspas ficam protegidas contra a transformação em minúsculas. As aspas permitem, também, incorporar espaços em branco ao identificador. Quando entre as aspas, um par de aspas é reduzido a uma única aspas no nome resultante. Por exemplo, `FOO"BAR"BAZ` é interpretado como `f00BARbaz`, e `"Um nome" "estranho"` se torna `Um nome "estranho"`.

A análise dos argumentos pára quando é encontrada outra contrabarra (não entre apóstrofes). Esta é considerada como sendo o início de um novo meta-comando. A seqüência especial `\\` (duas contrabarras) marca o fim dos argumentos e a continuação da análise dos comandos SQL, se existirem. Desta forma, os comandos SQL e `psql` podem ser livremente misturados na linha. Mas em nenhum caso os argumentos de um meta-comando podem continuar após o fim da linha.

Os seguintes meta-comandos estão definidos:

`\a`

Se o formato corrente de saída de tabela for desalinhado, troca para alinhado. Se não for desalinhado, define como desalinhado. Este comando é mantido para compatibilidade com as versões anteriores. Veja em `\pset` uma solução mais geral.

`\cd [ diretório ]`

Muda o diretório de trabalho corrente para `diretório`. Sem argumento, muda para o diretório `home` do usuário corrente.

**Dica:** Para ver o diretório de trabalho corrente deve ser usado `\!pwd`.

`\C [ título ]`

Define o título de todas as tabelas mostradas como resultado de uma consulta, ou remove a definição deste título. Este comando equivale a `\pset title título` (O nome deste comando deriva de “caption” (título), porque anteriormente só era usado para definir título em uma tabela HTML).

```
\connect (ou \c) [ nome_do_banco_de_dados [ nome_do_usuario ] ]
```

Estabelece a conexão com um banco de dados novo e/ou com um usuário novo. A conexão anterior é fechada. Se o *nome\_do\_banco\_de\_dados* for - (hífen), então é assumido o banco de dados corrente.

Se o *nome\_do\_usuario* for omitido, então o nome do usuário corrente é utilizado.

Como regra especial, o `\connect` sem nenhum argumento conecta ao banco de dados padrão com o usuário padrão (da mesma forma que aconteceria se o `psql` fosse iniciado sem argumentos).

Se a tentativa de conexão não for bem-sucedida (nome de usuário errado, acesso negado, etc.), a conexão anterior será mantida se, e somente se, o `psql` estiver no modo interativo. Quando estiver executando um script não interativo, o processamento será interrompido imediatamente com erro. Esta distinção foi escolhida por ser mais conveniente para o usuário, que digita menos, e como um mecanismo de segurança, impedindo os scripts de atuarem acidentalmente no banco de dados errado.

```
\copy tabela [ ( lista_de_colunas ) ] { from | to } { nome_do_arquivo | stdin | stdout |
pstdin | pstdout } [ with ] [ oids ] [ delimiter [ as ] 'caractere' ] [ null [ as ]
'string' ] [ csv [ quote [ as ] 'caractere' ] [ escape [ as ] 'caractere' ] [ force quote
lista_de_colunas ] [ force not null lista_de_colunas ] ]
```

Executa uma cópia pelo cliente. Esta é uma operação que executa o comando *COPY* do SQL, mas em vez do servidor ler ou escrever no arquivo especificado, o `psql` lê ou escreve no arquivo e roteia os dados entre o servidor e o sistema de arquivos local. Isto significa que a acessibilidade ao arquivo e os privilégios são do usuário local, e não do servidor, e que não há necessidade dos privilégios do superusuário.

A sintaxe deste comando é semelhante à sintaxe do comando *COPY* do SQL (Consulte sua descrição para obter detalhes). Deve ser observado que, por isso, são aplicadas regras especiais de análise ao comando `\copy`. Em particular, não se aplicam as regras de substituição de variável e de escape de contrabarra (`\`).

`\copy tabela from stdin / stdout` lê/escreve baseado na entrada e saída do comando, respectivamente. Todas as linhas são lidas a partir da mesma fonte que emitiu o comando, continuando até ser encontrado um `\.`, ou o fluxo chegar ao EOF. A saída é enviada para o mesmo local de saída do comando. Para ler/escrever da entrada ou saída padrão do `psql`, deve ser utilizado `pstdin` or `pstdout`. Esta opção é útil para carregar tabelas com dados contidos dentro do arquivo de script SQL.

**Dica:** Esta operação não é tão eficiente quanto o comando *COPY* do SQL, porque todos os dados passam através da conexão cliente/servidor. Para uma grande quantidade de dados, o comando SQL pode ser preferível.

O exemplo abaixo, executado no Windows, cria uma tabela e carrega os seus dados a partir de um arquivo de script que contém, no mesmo arquivo, tanto os comandos de definição de dados quanto os dados a serem carregados na tabela. Abaixo está mostrado o arquivo de script: <sup>2</sup>

```
-- Arquivo: f:\copy.sql
CREATE TEMPORARY TABLE t1 (c1 integer, c2 text);
\COPY t1 (c1,c2) FROM stdin WITH DELIMITER '|'
1|'um'
2|\N
3|'três'
\
\PSET NULL '(nulo)'
SELECT * FROM t1;
DROP TABLE t1;
```

e abaixo está mostrada a execução deste script:

```
=> \!chcp 1252
Active code page: 1252
=> \i 'f:\\copy.sql'
CREATE TABLE
Null display is "(nulo)".
```

```

c1 | c2
----+-----
 1 | 'um'
 2 | (nulo)
 3 | 'três'
(3 linhas)
DROP TABLE

```

\copyright

Mostra os termos da distribuição e dos direitos autorais do PostgreSQL.

\d [ *padrão* ]

\d+ [ *padrão* ]

Para cada relação (tabela, visão, índice ou seqüência) correspondendo ao *padrão*, mostra todas as colunas, seus tipos, o espaço de tabelas (se não for o padrão) e os atributos especiais como NOT NULL ou valor padrão, se houver. Os índices, restrições, regras e gatilhos associados também são mostrados, assim como a definição da visão se a relação for uma visão (A “Correspondência com padrão” é definida abaixo).

A forma \d+ do comando é idêntica, exceto por mostrar informações adicionais: são mostrados todos os comentários associados às colunas da tabela, e também a presença de OIDs na tabela.

**Nota:** Se \d for utilizado sem o argumento *padrão*, torna-se equivalente a \dtvs, mostrando a lista de todas as tabelas, visões e seqüências. Isto é puramente uma medida de conveniência.

\da [ *padrão* ]

Lista todas as funções de agregação disponíveis, junto com o tipo de dado com que operam. Se for especificado o *padrão*, somente são mostradas as agregações cujos nomes correspondem ao padrão.

\db [ *padrão* ]

\db+ [ *padrão* ]

Lista todos os espaços de tabela disponíveis. Se for especificado o *padrão*, somente são mostrados os espaços de tabela cujos nomes correspondem ao padrão. Se for anexado + ao nome do comando, é listado cada objeto juntamente com suas permissões associadas.

\dc [ *padrão* ]

Lista todas as conversões entre codificações de conjunto de caracteres disponíveis. Se for especificado o *padrão*, somente são mostradas as conversões cujos nomes correspondem ao padrão.

\dC

Mostra todas as conversões de tipo (cast) disponíveis.

\dd [ *padrão* ]

Mostra a descrição dos objetos cujos nomes correspondem ao *padrão*, ou todos os objetos visíveis se nenhum argumento for especificado. Nos dois casos somente são listados os objetos que possuem uma descrição (“Objeto” compreende agregações, funções, operadores, tipos, relações [tabelas, visões, índices, seqüências e objetos grandes], regras e gatilhos). Por exemplo:

=> \dd version

```

                        Object descriptions
 Schema | Name | Object | Description
-----+-----+-----+-----
pg_catalog | version | function | PostgreSQL version string
(1 linha)

```

As descrições dos objetos podem ser criadas pelo comando *COMMENT* do SQL.

\dD [ *padrão* ]

Lista todos os domínios disponíveis. Se for especificado o *padrão*, somente são mostrados os domínios cujos nomes correspondem ao padrão.

`\df [ padrão ]`

`\df+ [ padrão ]`

Lista as funções disponíveis, junto com o tipo de dado de seus argumentos e do valor retornado. Se for especificado o *padrão*, somente são mostradas as funções cujos nomes correspondem ao padrão. Se for usada a forma `\df+`, são mostradas informações adicionais sobre cada função, incluindo a linguagem e a descrição.

**Nota:** Para procurar por funções recebendo argumentos ou retornando valores de um determinado tipo, deve ser utilizada a capacidade de procurar do paginador, ou rolar através da saída do `\df`.

Para reduzir a desordem, o `\df` não mostra as funções com tipo de dado I/O. Isto é implementado ignorando as funções que recebem ou retornam o tipo `cstring`.

`\dg [ padrão ]`

Lista todos os grupos de bancos de dados. Se for especificado o *padrão*, somente são mostrados os grupos cujos nomes correspondem ao padrão.

`\distvs [ padrão ]`

Este não é, na verdade, o nome do comando: As letras *i*, *s*, *t*, *v*, *S* correspondem a índice, seqüência, tabela, visão e tabela do sistema, respectivamente. Pode ser especificada qualquer uma ou todas as letras, em qualquer ordem, para obter a listagem de todos os objetos correspondentes. A letra *S* restringe a listagem aos objetos do sistema; sem o *S*, somente são mostrados os objetos que não são do sistema. Se for anexado o caractere *+* ao nome do comando, cada objeto é listado junto com sua descrição associada, se houver.

Se for especificado o *padrão*, somente são mostrados os objetos cujos nomes correspondem ao padrão.

`\dl`

Este é um outro nome para `\lo_list`, que mostra a lista dos objetos grandes.

`\dn [ padrão ]`

`\dn+ [ padrão ]`

Lista todos os esquemas (espaços de nomes) disponíveis. Se for especificado o *padrão* (uma expressão regular), somente são mostrados os esquemas cujos nomes correspondem ao padrão. Os esquemas temporários não-locais são suprimidos. Se for anexado o caractere *+* ao nome do comando, cada objeto é listado junto com sua descrição e permissões associadas, se houver.

`\do [ padrão ]`

Lista os operadores disponíveis junto com o tipo de seus operandos e do valor retornado. Se for especificado o *padrão*, somente são mostrados os operadores cujos nomes correspondem ao padrão.

`\dp [ padrão ]`

Produz uma lista contendo todas as tabelas, visões e seqüências disponíveis, junto com seus privilégios de acesso associados. Se for especificado o *padrão*, somente são mostradas as tabelas, visões e seqüências cujos nomes correspondem ao padrão.

Os comandos GRANT e REVOKE são utilizados para definir os privilégios de acesso. Consulte o comando GRANT para obter informações adicionais.

`\dT [ padrão ]`

`\dT+ [ padrão ]`

Mostra todos os tipos de dado, ou somente aqueles que correspondem ao *padrão*. A forma do comando `\dT+` mostra informações adicionais.

`\du [ padrão ]`

Lista todos os usuários do banco de dados, ou somente aqueles que correspondem ao *padrão*.

`\edit (ou \e) [ nome_do_arquivo ]`

Se for especificado o *nome\_do\_arquivo*, o arquivo é editado; após o fim da edição (fechar o editor), o conteúdo do arquivo é copiado para o *buffer* de comando. Se não for fornecido nenhum argumento, o *buffer* de comando corrente é copiado para um arquivo temporário, que é editado de forma idêntica.

O novo `buffer` de comando é então analisado novamente, de acordo com as regras normais do `psql`, onde todo o `buffer` é tratado como sendo uma única linha (Portanto, não podem ser gerados scripts dessa forma. Use o comando `\i` para isso). Significa também que, se o comando terminar por (ou contiver) um ponto-e-vírgula, será executado imediatamente. Se não contiver, apenas permanecerá aguardando no `buffer` de comando.

**Dica:** O `psql` procura nas variáveis de ambiente `PSQL_EDITOR`, `EDITOR` e `VISUAL` (nesta ordem) o editor a ser usado. Se nenhuma delas estiver definida, então é usado `/bin/vi` nos sistemas Unix, ou o `notepad.exe` no Windows.

`\echo texto [ ... ]`

Envia os argumentos para a saída padrão, separados por um espaço e seguido por um caractere de nova-linha, o que pode ser útil para intercalar informações na saída dos scripts. Por exemplo:

```
=> \echo `date`
Seg Feb 21 09:17:00 BRT 2005
```

Se o primeiro argumento for `-n` (não entre apóstrofes) não é enviado o caractere de nova-linha final.

**Dica:** Se for usado o comando `\o` para redirecionar a saída do comando, talvez seja preferível utilizar `\qecho` em vez deste comando.

`\encoding [ codificação ]`

Define a codificação do conjunto de caracteres do cliente. Sem argumento, este comando mostra a codificação corrente. Por exemplo:

```
=> \encoding
LATIN1
```

`\f [ cadeia_de_caracteres ]`

Define o separador de campos para a saída de comando desalinhada. O padrão é a barra vertical (`|`). Veja também em `\pset` uma forma genérica para definir as opções de saída.

`\g [ { nome_do_arquivo | | comando } ]`

Envia o `buffer` de entrada de comando corrente para o servidor e, opcionalmente, armazena a saída do comando em `nome_do_arquivo`, ou envia a saída para outro interpretador de comandos do Unix executando o `comando`. Um `\g` puro e simples é virtualmente equivalente ao ponto-e-vírgula. Um `\g` com argumento é uma alternativa “de uma única vez” para o comando `\o`.

O exemplo abaixo coloca a saída do comando `SELECT` na área de edição do editor `gvim`:<sup>3</sup>

```
=> SELECT datname FROM pg_database \g | gvim -
=> Vim: Reading from stdin...
```

`\help (ou \h) [ comando ]`

Fornecer ajuda de sintaxe para o comando SQL especificado. Senão for especificado o `comando`, então o `psql` lista todos os comandos para os quais existe ajuda de sintaxe disponível. Se o `comando` for um asterisco (“\*”), então é mostrada a ajuda de sintaxe para todos os comandos SQL.

**Nota:** Para simplificar a digitação, os comandos compostos por várias palavras não necessitam estar entre apóstrofes. Portanto, pode ser digitado `\help alter table`.

`\H`

Habilita o formato de saída de comando HTML. Se o formato HTML já estiver habilitado, retorna ao formato de texto alinhado padrão. Este comando existe por compatibilidade e comodidade, mas deve ser visto em `\pset` as definições de outras opções de saída.

`\i nome_do_arquivo`

Lê a entrada no arquivo `nome_do_arquivo`, e executa como se tivesse sido digitada pelo teclado.

**Nota:** Se for desejado ver as linhas na tela à medida que são lidas, deve ser definida a variável `ECHO` como `all`.

`\l` (ou `\list`)  
`\l+` (ou `\list+`)

Lista o nome, dono e codificação do conjunto de caracteres de todos os bancos de dados do servidor. Se for adicionado o caractere + ao nome do comando, também são mostradas todas as descrições dos bancos de dados.

`\lo_export loid nome_do_arquivo`

Lê no banco de dados o objeto grande com OID igual a *loid*, e escreve em *nome\_do\_arquivo*. Deve ser observado que isto é sutilmente diferente da função do servidor `lo_export`, que atua com a permissão do usuário como o qual o servidor de banco de dados está executando, e no sistema de arquivos do servidor.

**Dica:** Use `\lo_list` para descobrir os OIDs dos objetos grandes.

`\lo_import nome_do_arquivo [ comentário ]`

Armazena o arquivo em um objeto grande do PostgreSQL. Opcionalmente, associa o comentário fornecido ao objeto. Exemplo:

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'uma fotografia minha'
lo_import 152801
```

A resposta indica que o objeto grande recebeu o identificador de objeto 152801, que deve ser lembrado para acessar o objeto novamente. Por esta razão, recomenda-se associar sempre um comentário inteligível a cada objeto. Estes podem ser vistos utilizando o comando `\lo_list`.

Deve ser observado que este comando é sutilmente diferente da função `lo_import` do servidor, porque atua como o usuário local no sistema de arquivos local, em vez do usuário do servidor no sistema de arquivos do servidor.

`\lo_list`

Mostra a lista de todos os objetos grandes do PostgreSQL armazenados neste instante no banco de dados, junto com os comentários fornecidos para os mesmos.

`\lo_unlink loid`

Remove do banco de dados o objeto grande com o OID igual a *loid*.

**Dica:** Use `\lo_list` para descobrir os OIDs dos objetos grandes.

`\o [ {nome_do_arquivo | |comando} ]`

Salva os resultados dos próximos comandos no arquivo *nome\_do\_arquivo*, ou envia os próximos resultados para um outro interpretador de comandos do Unix para executar o *comando*. Se não for especificado nenhum argumento, a saída do comando será redefinida para a saída padrão.

O exemplo abaixo coloca a saída dos comandos `SELECT` e `\qecho` na área de edição do editor `gvim`:<sup>4</sup>

```
=> \o | gvim -
=> Vim: Reading from stdin...
=> \qecho Bancos de Dados
=> SELECT datname FROM pg_database;
=> \qecho Linguagens
=> SELECT lanname FROM pg_language;
=> \o
```

Os “resultados dos comandos” incluem todas as tabelas, respostas dos comandos e notificações recebidas do servidor de banco de dados, assim como a saída de vários comandos de contrabarra que consultam o banco de dados (como o `\d`), mas não as mensagens de erro.

**Dica:** Para intercalar saída de texto entre os resultados dos comandos deve ser utilizado `\qecho`.

`\p`

Envia o buffer de comando corrente para a saída padrão.



```
\pset parâmetro [ valor ]
```

Este comando define opções que afetam a saída das tabelas de resultado dos comandos. O *parâmetro* indica qual opção será definida. A semântica do *valor* depende do parâmetro.

As opções de exibição ajustáveis são:

**format**

Define o formato de saída como `unaligned` (desalinhado), `aligned` (alinhado), `html` ou `latex`. São permitidas abreviações únicas (O que significa que basta uma letra).

O modo “unaligned” escreve todas as colunas de uma linha em uma única linha, separadas pelo separador de campos ativo corrente. Pretende-se com isso criar uma saída que sirva de entrada para outros programas (separada por tabulação, vírgula, etc.). O modo “aligned” é a saída de texto padrão, inteligível e agradavelmente formatada. Os modos “HTML” e “LaTeX” produzem tabelas feitas para serem incluídas em documentos usando a linguagem de marcação correspondente. Não são documentos completos! (Isto não é tão problemático no HTML, mas no LaTeX deve haver um invólucro completo do documento).

**border**

O segundo argumento deve ser um número. Em geral, quanto maior o número mais bordas e linhas a tabela terá, mas isto depende do formato. No modo HTML será traduzido diretamente para o atributo `border=...`, nos demais modos só fazem sentido os valores: 0 (sem borda); 1 (linhas divisórias internas); e 2 (moldura da tabela).

**expanded (ou x)**

Alterna entre os formatos regular e expandido. Quando o formato expandido está habilitado, todas as saídas possuem duas colunas, com o nome da coluna à esquerda e o dado à direita. Este modo é útil quando os dados não cabem na tela no modo normal “horizontal”.

O modo expandido é suportado por todos os quatro formatos de saída.

**null**

O segundo argumento é a cadeia de caracteres a ser mostrada sempre que a coluna for nula. O padrão é não mostrar nada, que pode ser facilmente confundido com, por exemplo, uma cadeia de caracteres vazia. Portanto, pode-se preferir escrever `\pset null '(nulo)'`.

**fieldsep**

Especifica o separador de campos a ser utilizado no modo de saída desalinhado. Desta forma pode ser criada, por exemplo, uma saída separada por tabulação ou por vírgula, que os outros programas podem preferir. Para definir o caractere de tabulação como separador de campos deve ser usado `\pset fieldsep '\t'`. O separador de campos padrão é `|` (a barra vertical).

**footer**

Alterna a exibição do rodapé padrão (`x` linhas).

**recordsep**

Especifica o separador de registro (linha) a ser usado no modo de saída desalinhado. O padrão é o caractere de nova-linha.

**tuples\_only (ou t)**

Alterna entre mostrar somente as tuplas e mostrar tudo. O modo mostrar tudo pode mostrar informações adicionais como os cabeçalhos das colunas, títulos e vários rodapés. No modo somente-tuplas são mostrados apenas os dados da tabela.

**title [ texto ]**

Define o título das próximas tabelas mostradas. Pode ser usado para colocar textos descritivos na saída. Se não for fornecido nenhum argumento, o título é removido.

`tableattr` (ou `T`) [ *texto* ]

Permite especificar qualquer atributo a ser colocado dentro da marca `table` do HTML. Estes atributos podem ser, por exemplo, `cellpadding` ou `bgcolor`. Deve ser observado que, provavelmente, não será desejado especificar `border` aqui, porque isto já é tratado pelo `\pset border`.

`pager`

Controla o uso do paginador para comandos e para a saída da ajuda do psql. Se a variável de ambiente `PAGER` estiver definida, a saída será enviada para o programa especificado, senão é utilizado o padrão dependente da plataforma (como o `more`).

Quando o paginador está desabilitado, a paginação não é feita. Quando o paginador está habilitado, a paginação é feita somente quando for apropriado, ou seja, quando a saída é para um terminal e não cabe na tela (O psql não realiza um trabalho perfeito ao avaliar quando o paginador deve ser utilizado). `\pset pager` habilita e desabilita o paginador. O paginador também pode ser definido como `always`, o que faz o paginador ser utilizado sempre.

Ilustrações mostrando como se parecem estes formatos diferentes podem ser vistas na seção *Exemplos*.

**Dica:** Existem vários comandos abreviados para o `\pset`. Veja `\a`, `\C`, `\H`, `\t`, `\T` e `\x`.

**Nota:** É errado chamar o `\pset` sem argumentos. No futuro, esta chamada deverá mostrar o status corrente de todas as opções de exibição.

`\q`

Sair do programa psql.

`\qecho texto [ ... ]`

Este comando é idêntico a `\echo`, exceto que toda a saída é escrita no canal de saída de comando, conforme definido por `\o`.

`\r`

Redefine (limpa) o `buffer` de comando.

`\s [ nome_do_arquivo ]`

Mostra ou salva o histórico da linha de comando em `nome_do_arquivo`. Se for omitido o `nome_do_arquivo`, o histórico é escrito na saída padrão. Esta opção somente estará disponível se o psql estiver configurado para usar a biblioteca Readline do GNU.

**Nota:** Na versão corrente não é mais necessário salvar o histórico de comandos, porque isso é feito automaticamente ao término do programa. O histórico também é carregado, automaticamente, toda vez que o psql inicia.

`\set [ nome [ valor [ ... ] ] ]`

Define a variável interna `nome` com o `valor` ou, se for fornecido mais de um valor, com a concatenação de todos os valores. Se não for fornecido o segundo argumento a variável somente é definida, sem nenhum valor. Para remover a definição da variável deve ser usado o comando `\unset`.

Nomes de variáveis válidos podem conter letras, dígitos e sublinhados (`_`). Consulte a seção *Variáveis* abaixo para obter detalhes.

Embora possa ser definida qualquer variável como qualquer coisa desejada, o psql trata várias variáveis como sendo especiais. Elas estão documentadas na seção sobre variáveis.

**Nota:** Este comando é totalmente distinto do comando `SET` do SQL.

`\t`

Alterna a exibição do cabeçalho contendo o nome das colunas e do rodapé contendo o número de linhas. Este comando equivale a `\pset tuples_only`, sendo fornecido por conveniência.

`\T opções_de_tabela`

Permite especificar atributos a serem colocados na marca `table` no modo de saída tabular HTML. Este comando equivale a `\pset tableattr opções_de_tabela`.

`\timing`

Alterna a exibição de quanto tempo cada comando SQL demora, em milissegundos.

`\w {nome_do_arquivo | /comando}`

Escreve o buffer de comando corrente no arquivo *nome\_do\_arquivo*, ou envia para o comando Unix *comando* através de um pipe.

O exemplo abaixo utiliza o comando `\w` para executar dois comandos do sistema operacional: <sup>5</sup>

```
=> \w | date
Dom Feb 20 07:55:08 BRT 2005
=> \w | pwd
/root
```

`\x`

Alterna o modo de formatação de tabela estendido. Como tal equivale a `\pset expanded`.

`\z [ padrão ]`

Produz uma lista contendo todas as tabelas, visões e seqüências disponíveis, junto com seus privilégios de acesso associados. Se for especificado o *padrão*, somente são mostradas as tabelas, visões e seqüências cujos nomes correspondem ao padrão.

Os comandos GRANT e REVOKE são utilizados para definir os privilégios de acesso. Consulte o comando GRANT para obter informações adicionais.

Este comando é um outro nome para o comando `\dp` (“display privileges”).

`\! [ comando ]`

Abre um outro interpretador de comandos do Unix, ou executa o comando Unix *comando*. Os argumentos não são mais interpretados, sendo enviados para o interpretador de comandos como estão.

O exemplo abaixo utiliza o meta-comando `\!` para executar dois comandos do sistema operacional Unix: <sup>6</sup>

```
=> \!date
Dom Feb 20 08:04:41 BRT 2005
=> \!pwd
/root
```

`\?`

Mostra informação de ajuda para os comandos de contrabarra (“”).

Vários comandos `\d` aceitam como parâmetro um *padrão* para especificar os nomes dos objetos a serem mostrados. O `*` significa “qualquer seqüência de caracteres” e `?` significa “qualquer um único caractere” (Esta notação é semelhante a do padrão para nomes de arquivos do interpretador de comandos do Unix). Os usuários avançados também podem utilizar a notação das expressões regulares, como as classes de caracteres; por exemplo `[0-9]` correspondendo a “qualquer dígito”. Para fazer qualquer um desses caracteres de correspondência com padrão ser interpretado literalmente, deve-se colocá-lo entre aspas.

Um padrão contendo um ponto (não entre aspas) é interpretado como um padrão de nome de esquema seguido por um padrão de nome de objeto. Por exemplo, `\dt foo*.bar*` mostra todas as tabelas nos esquemas cujos nomes começam por `foo`, e cujos nomes de tabela começam por `bar`. Se não houver nenhum ponto, então o padrão corresponde apenas aos objetos visíveis no caminho de procura de esquemas corrente. Por exemplo: (N. do T.)

```
=> \dt info*.*sizing*
```

```

              Lista de relações
  Esquema      | Nome                | Tipo | Dono
-----+-----+-----+-----
information_schema | sql_sizing          | tabela | postgres
information_schema | sql_sizing_profiles | tabela | postgres
(2 linhas)

```

```
=> \dt 'info*.*sizing*'
```

```

              Lista de relações
  Esquema      | Nome                | Tipo | Dono
-----+-----+-----+-----
information_schema | sql_sizing          | tabela | postgres
information_schema | sql_sizing_profiles | tabela | postgres
(2 linhas)

```

```
=> \dt "info*.*sizing"
```

Não foi encontrada relação correspondente.

Sempre que o parâmetro *padrão* é omitido, o comando `\d` mostra todos os objetos visíveis no caminho de procura de esquemas corrente. Para mostrar todos os objetos do banco de dados, deve ser usado o padrão `*.*`.

## Funcionalidades avançadas

### Variáveis

O psql fornece uma funcionalidade de substituição de variáveis semelhante a dos interpretadores de comando do Unix. As variáveis são simplesmente pares nome/valor, onde o valor pode ser qualquer cadeia de caracteres de qualquer comprimento. Para definir as variáveis é utilizado o meta-comando do psql `\set`:

```
testdb=> \set foo bar
```

define a variável `foo` com o valor `bar`. Para acessar o conteúdo da variável deve-se preceder seu nome por dois-pontos (:), e usá-lo como argumento de qualquer comando de contrabarra:

```
testdb=> \echo :foo
bar
```

**Nota:** Os argumentos do `\set` estão sujeitos às mesmas regras de substituição de qualquer outro comando. Portanto, podem ser construídas referências interessantes como `\set :foo 'something'` e obter “soft links” ou “variable variables” do Perl e do PHP, respectivamente. Desafortunadamente (ou afortunadamente?), não existe nenhuma forma de fazer algo útil com estas construções. Por outro lado, `\set bar :foo` é uma forma perfeitamente válida de copiar uma variável.

Se `\set` for chamado sem o segundo argumento, a variável é definida com uma cadeia de caracteres vazia como valor. Para remover a definição (ou excluir) a variável, deve ser utilizado o comando `\unset`.

Os nomes das variáveis internas do psql podem ser formados por letras, números e sublinhados em qualquer ordem e número. Algumas destas variáveis recebem tratamento especial pelo psql. Denotam determinadas configurações de opção que podem ser mudadas em tempo de execução alterando o valor da variável, ou representam algum estado do aplicativo. Embora seja possível usar estas variáveis para qualquer outra finalidade isto não é recomendado, porque o comportamento do programa pode ficar muito estranho, muito rapidamente. Por convenção, todas as variáveis com tratamento especial possuem todas as letras maiúsculas (e possivelmente números e sublinhados). Para garantir a máxima compatibilidade futura, evite usar estes nomes de variáveis para as suas próprias finalidades. Abaixo segue a lista de todas as variáveis com tratamento especial.

### AUTOCOMMIT

Quando está `on` (o padrão), cada comando SQL é automaticamente efetivado ao terminar bem-sucedido. Neste modo, para adiar a efetivação deve ser entrado um comando SQL `BEGIN` ou `START TRANSACTION`. Quando está `off`, ou não definido, os comandos SQL não são efetivados até ser emitido explicitamente o comando `COMMIT` ou `END`. O modo “autocommit-off” opera emitindo um comando `BEGIN`, implícito, logo antes de qualquer comando que não esteja

dentro de um bloco de transação, e não seja o próprio `BEGIN` ou outro comando de controle de transação, nem um comando que não possa ser executado dentro de uma transação, (como o `VACUUM`).

**Nota:** No modo “autocommit-off” toda transação que não for bem-sucedida deve ser explicitamente abandonada entrando com `ABORT` ou `ROLLBACK`. Também tenha em mente que, se a sessão for terminada sem a transação ser efetivada, o trabalho será perdido.

**Nota:** O modo “autocommit-on” é o comportamento tradicional do PostgreSQL, mas o modo “autocommit-off” é mais próximo da especificação SQL. Se for preferido o modo “autocommit-off”, este pode ser definido no arquivo `psqlrc` global do sistema, ou no arquivo `~/.psqlrc` do usuário.

O exemplo abaixo mostra o valor da variável `AUTOCOMMIT`: <sup>7</sup>

```
=> \echo :AUTOCOMMIT
on
```

`DBNAME`

O nome do banco de dados que se está conectado no momento. Definida toda vez que é feita a conexão com um banco de dados (inclusive na inicialização do programa), mas a definição pode ser removida.

O exemplo abaixo mostra o valor da variável `DBNAME`: <sup>8</sup>

```
=> \echo :DBNAME
template1
```

`ECHO`

Se for definida como “all”, todas as linhas entradas pelo teclado, ou do script, são escritas na saída padrão antes de serem analisadas ou executadas. Para selecionar este comportamento na inicialização do programa, deve ser usada a chave `-a`. Se for definida como “queries”, o psql simplesmente mostra todos os comandos à medida que são enviados para o servidor. A chave para isto é `-e`.

`ECHO_HIDDEN`

Quando esta variável está definida, e um comando de contrabarra consulta o banco de dados, primeiro a consulta é mostrada. Por este meio pode-se estudar a parte interna do PostgreSQL e oferecer funcionalidades semelhantes nos próprios programas (Para selecionar este comportamento na inicialização do programa, deve ser usada a chave `-E`). Se a variável for definida com o valor “noexec” os comandos são apenas mostrados, não são enviados para o servidor para serem executados.

O exemplo abaixo define a variável `ECHO_HIDDEN` e executa o meta-comando `\dn`: <sup>9</sup>

```
=> \set ECHO_HIDDEN
=> \dn
***** QUERY *****
SELECT n.nspname AS "Nome",
       u.username AS "Dono"
FROM pg_catalog.pg_namespace n LEFT JOIN pg_catalog.pg_user u
     ON n.nspowner=u.usesysid
WHERE  (n.nspname NOT LIKE 'pg\\_temp\\_%' OR
       n.nspname = (pg_catalog.current_schemas(true))[1])
ORDER BY 1;
*****
```

```

      Lista dos esquemas
      Nome          |  Dono
-----+-----
information_schema | postgres
pg_catalog         | postgres
pg_toast           | postgres
public             | postgres
(4 linhas)
```

## ENCODING

A codificação corrente do conjunto de caracteres do cliente.

O exemplo abaixo mostra o valor da variável `ENCODING`:<sup>10</sup>

```
=> \echo :ENCODING
LATIN1
```

## HISTCONTROL

Se esta variável estiver definida como “`ignoreSPACE`”, as linhas começando por espaço não são guardadas na lista de histórico. Se estiver definida como “`ignoreDUPS`”, as linhas idênticas à linha anterior do histórico não são guardadas. O valor “`ignoreBOTH`” combina estas duas opções. Se não estiver definida, ou se estiver definida com um valor diferente destes acima, todas as linhas lidas no modo interativo são guardadas na lista de histórico.

**Nota:** Esta funcionalidade foi desavergonhadamente plagiada do Bash.

## HISTSIZE

O número de comandos a serem guardados no histórico de comandos. O valor padrão é 500.

**Nota:** Esta funcionalidade foi desavergonhadamente plagiada do Bash.

## HOST

O hospedeiro do servidor de banco de dados ao qual se está conectado. Definida toda vez que se conecta com o banco de dados (inclusive na inicialização do programa), mas a definição pode ser removida.

## IGNOREEOF

Se não estiver definida, o envio do caractere EOF (geralmente **Control+D**) para uma sessão interativa do psql termina o aplicativo. Se estiver definida com um valor numérico, é ignorada esta quantidade de caracteres EOF antes do aplicativo terminar. Se a variável estiver definida, mas não tiver um valor numérico, o padrão é 10.

**Nota:** Esta funcionalidade foi desavergonhadamente plagiada do Bash.

## LASTOID

O valor do último `OID` afetado, conforme retornado por um comando `INSERT` ou `lo_insert`. Somente há garantia desta variável ser válida até ser mostrado o resultado do próximo comando SQL.

## ON\_ERROR\_STOP

Por padrão, se um script não-interativo encontrar algum erro, como um comando SQL ou um meta-comando interno mal formado, o processamento continua. Este tem sido o comportamento tradicional do psql, mas algumas vezes não é o desejado. Se esta variável estiver definida, o processamento do script terminará imediatamente. Se o script foi chamado por outro script este terminará da mesma maneira. Se o script mais externo não foi chamado por uma sessão interativa do psql, mas usando a opção `-f`, o psql retorna o código de erro 3, para distinguir este caso das condições de erro fatal (código de erro 1).

## PORT

A porta do servidor de banco de dados ao qual se está conectado. Definida toda vez que se conecta com o banco de dados (inclusive na inicialização do programa), mas a definição pode ser removida.

## PROMPT1

## PROMPT2

## PROMPT3

Especificam como os `prompts` emitidos pelo psql devem se parecer. Consulte *Prompt* abaixo.

## QUIET

Esta variável equivale à opção de linha de comando `-q`. Provavelmente não tem muita utilidade no modo interativo.

## SINGLELINE

Esta variável equivale à opção de linha de comando `-s`.

## SINGLESTEP

Esta variável equivale à opção de linha de comando `-s`.

## USER

O usuário do banco de dados como o qual se está conectado. Definida toda vez que é feita a conexão com um banco de dados (inclusive na inicialização do programa), mas a definição pode ser removida.

## VERBOSITY

Esta variável pode ser definida com os valores `default`, `verbose` ou `terse` (sucinto), para controlar a verbosidade dos relatórios de erro.

*Interpolação SQL*

Uma funcionalidade adicional útil das variáveis do psql é poderem ser substituídas (“interpoladas”) dentro de comandos SQL regulares. Novamente a sintaxe é colocar dois-pontos (:) como prefixo do nome da variável.

```
testdb=> \set foo 'minha_tabela'
testdb=> SELECT * FROM :foo;
```

faria então a consulta à tabela `minha_tabela`. O valor da variável é copiado literalmente podendo, portanto, conter apóstrofes não balanceados ou comandos de contrabarra. Deve-se ter certeza que faz sentido onde é colocada. Não é realizada a interpolação de variáveis dentro de entidades SQL entre aspas ou apóstrofes. Por exemplo: (N. do T.)

```
testdb=> \set tabela 'pg_catalog.pg_user'
testdb=> \echo :tabela
pg_catalog.pg_user
testdb=> \echo ':tabela'
:tabela
testdb=> \echo ":tabela"
":tabela"
testdb=> SELECT username FROM :tabela WHERE usesysid=1;
username
-----
postgres
(1 linha)
testdb=> SELECT username FROM ':tabela' WHERE usesysid=1;
ERRO:  erro de sintaxe em ou próximo de "':tabela'" no caractere 21
LINHA 1: SELECT username FROM ':tabela' WHERE usesysid=1;
      ^
testdb=> SELECT username FROM ":tabela" WHERE usesysid=1;
ERRO:  a relação ":tabela" não existe
```

Uma aplicação comum desta funcionalidade é para fazer referência nos comandos subseqüentes ao último OID inserido, para construir um cenário de chave estrangeira. Outra utilização possível deste mecanismo é para copiar o conteúdo de um arquivo para uma coluna de uma tabela. Primeiro deve ser carregado o arquivo na variável, e depois proceder conforme mostrado.

```
testdb=> \set conteudo '\`cat meu_arquivo.txt` \'
testdb=> INSERT INTO minha_tabela VALUES (:conteudo);
```

Um problema possível com esta abordagem é que `meu_arquivo.txt` pode conter apóstrofes, que devem ser precedidos por contrabarra para não causarem erro de sintaxe quando a segunda linha for processada, o que pode ser feito por meio do programa `sed`:

```
testdb=> \set conteudo '\`sed -e "s/'/\\\\'/g" < meu_arquivo.txt` \'
```

Deve ser observado o número correto de contabarras (6)! Isto funciona da seguinte maneira: Após o psql ter analisado esta linha, enviará `sed -e "s/'/\\\\'/g" < meu_arquivo.txt` para o interpretador de comandos, que fará suas próprias atividades dentro das aspas e executará o `sed` com os argumentos `-e` e `s/'/\\\\'/g`. Quando o `sed` fizer a análise substituirá as duas contrabarras por uma única e, então, fará a substituição. Talvez em algum ponto tenha se pensado ser ótimo todos os comandos Unix utilizarem o mesmo caractere de escape (escape character). Isto tudo ainda ignora o

fato de ter que colocar contrabarra na frente de contrabarra também, porque as constantes textos do SQL também estão sujeitas a certas interpretações. Neste caso é melhor preparar o arquivo externamente.

Uma vez que os dois-pontos podem aparecer legalmente nos comandos SQL, a seguinte regra se aplica: a seqüência de caracteres “:nome” não é modificada a menos que “nome” seja o nome de uma variável atualmente definida. Sempre pode ser feito o escape dos dois-pontos com uma contrabarra para protegê-lo da substituição (A sintaxe dos dois-pontos para as variáveis é padrão SQL para linguagens de comandos incorporados, tal como ECPG. A sintaxe de dois-pontos para “faixa de matriz” e “conversão de tipo” são extensões do PostgreSQL, daí o conflito).

### Prompt

Os prompts emitidos pelo psql podem ser personalizados conforme a preferência. As três variáveis PROMPT1, PROMPT2 e PROMPT3 contêm cadeias de caracteres e seqüências especiais de escape que descrevem a aparência do prompt. O prompt 1 é o prompt normal emitido quando o psql solicita um novo comando. O prompt 2 é emitido durante a entrada do comando quando mais entrada é aguardada, porque o comando não foi terminado por um ponto-e-vírgula, ou um apóstrofo não foi fechado. O prompt 3 é emitido quando se executa o comando COPY do SQL, e se espera que os valores das linhas sejam digitados no terminal.

O valor da variável de prompt selecionada é exibido literalmente, exceto quando um sinal de percentagem (%) é encontrado. Dependendo do caractere seguinte, certos outros textos são colocados em seu lugar. As substituições definidas são:

%M

O nome completo do hospedeiro do servidor de banco de dados (com o nome do domínio), ou [local] se a conexão for através de um soquete do domínio Unix, ou [local:/dir/nome], se o soquete do domínio Unix não estiver no local padrão da compilação.

%m

O nome, truncado no primeiro ponto, do hospedeiro do servidor de banco de dados, ou [local] se a conexão for através de um soquete do domínio Unix.

%>

O número da porta onde o servidor de banco de dados está atendendo.

%n

O nome do usuário da sessão de banco de dados (A expansão deste valor pode mudar durante a sessão de banco de dados como resultado do comando SET SESSION AUTHORIZATION.)

%/

O nome do banco de dados corrente.

%~

Como %/, mas a saída será “~” (til) se o banco de dados for o banco de dados padrão.

%#

Se o usuário da sessão for um superusuário do banco de dados então um #, senão > (A expansão deste valor pode mudar durante a sessão de banco de dados como resultado do comando SET SESSION AUTHORIZATION.)

%R

No prompt 1 normalmente “=”, mas “^” se estiver no modo linha-única, e “!” se a sessão estiver desconectada do banco de dados (o que pode acontecer se \connect não for bem-sucedido). No prompt 2 a seqüência é substituída por “-”, “\*”, apóstrofo, aspas ou “\$”, dependendo se o psql está aguardando mais entrada devido ao comando não ter terminado ainda, porque está dentro de um comentário /\* ... \*/ , ou porque está dentro de uma cadeia de caracteres envolta por apóstrofes, aspas ou cifrão. No prompt 3 a seqüência não se transforma em nada.

%x

Status da transação: uma cadeia de caracteres vazia quando não estiver dentro de um bloco de transação, ou \* quando estiver dentro de um bloco de transação, ou ! quando estiver em um bloco de transação que falhou, ou ? quando o estado da transação for indeterminado (por exemplo, porque não há conexão).



`%dígitos`

O caractere com o código numérico indicado é substituído. Se *dígitos* começar por 0x os demais caracteres são interpretados como hexadecimal; senão, se o primeiro dígito for 0, os dígitos são interpretados como octal; senão os dígitos são lidos como um número decimal.

`%:nome:`

O valor da variável do psql *nome*. Consulte a seção *Variáveis* para obter detalhes.

`%`comando``

A saída do *comando*, semelhante à substituição de “crase” normal.

`%[ ... %]`

Os `prompts` podem conter caracteres de controle do terminal como, por exemplo, para mudar a cor, fundo ou estilo do `prompt`. Para que as funcionalidades de edição de linha do Readline funcionem de forma apropriada, estes caracteres de controle não imprimíveis devem ser projetados como invisíveis envolvendo-os por `%[ e %]`. Podem existir vários pares iguais a este dentro do `prompt`. Por exemplo,

```
testdb=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%##] '
```

resulta em um `prompt` negrito (1;) amarelo-sobre-preto (33;40) em um terminal colorido compatível com o VT100.

Para inserir um sinal de percentagem no `prompt` deve ser escrito `%%`. Os `prompts` padrão são `'%/%R%# '` para os `prompts` 1 e 2, e `'>> '` para o `prompt` 3.

**Nota:** Esta funcionalidade foi desavergonhadamente plagiada do `tcsh`.

### Edição da linha de comando

O psql usa a biblioteca Readline para fornecer uma recuperação e edição de linha conveniente. O histórico é salvo automaticamente quando o psql termina, e é recarregado quando o psql inicia. Completar com a tecla de tabulação também é suportado, embora a lógica do ato de completar não pretenda ser a de um analisador SQL. Se por algum motivo não se gostar de completar com tabulação, pode-se desativar especificando isto no arquivo chamado `.inputrc` no diretório `home` do usuário:

```
$if psql
set disable-completion on
$endif
```

(Esta não é uma funcionalidade do psql, mas sim do Readline. Leia sua documentação para obter mais detalhes).

## Ambiente

PAGER

Se o resultado do comando não couber na tela, então é mostrado por meio deste comando. Os valores típicos são `more` e `less`. O padrão depende da plataforma. O uso de um paginador pode ser desabilitado por meio do comando `\pset`.

PGDATABASE

Banco de dados padrão para conectar.

PGHOST

PGPORT

PGUSER

Parâmetros de conexão padrão.

PSQL\_EDITOR

EDITOR

VISUAL

Editor utilizado pelo comando `\e`. As variáveis são examinadas na ordem listada; a primeira que estiver definida é utilizada.

## SHELL

Comando executado pelo meta-comando \!.

## TMPDIR

Diretório para armazenar os arquivos temporários. O padrão é /tmp.

## Arquivos

- Antes de iniciar, o psql tenta ler e executar os comandos do arquivo global do sistema `psqlrc`, e o arquivo do usuário `~/.psqlrc` (No Windows, o arquivo de inicialização do usuário chama-se `%APPDATA%\postgresql\psqlrc.conf`). Veja em `PREFIX/share/psqlrc.sample` as informações sobre como definir o arquivo global do sistema. Pode ser usado para configurar o cliente e o servidor conforme se deseje (usando os comandos `\set` e `SET`).
- Tanto o arquivo global do sistema, `psqlrc`, quanto o arquivo do usuário, `~/.psqlrc`, podem ser tornados específicos para uma versão anexando um hífen e o número da versão do PostgreSQL como, por exemplo, `~/.psqlrc-8.0.0`. O arquivo correspondendo à versão específica terá preferência sobre o arquivo que não é específico da versão.
- O histórico de linha de comando é armazenado no arquivo `~/.psql_history`, ou no arquivo `%APPDATA%\postgresql\psql_history` no Windows.

## Observações

- Nas versões iniciais, o psql permitia o primeiro argumento de um comando de contrabarra de uma única letra começar logo após o comando, sem o espaço separador. Por motivo de compatibilidade isto ainda é suportado de alguma forma, que não será explicada em detalhes porque seu uso é desencorajado, mas se forem recebidas mensagens estranhas tenha isso em mente. Por exemplo

```
testdb=> \foo
Field separator is "oo".
```

talvez não seja o esperado.

- O psql somente trabalha adequadamente com servidores da mesma versão. Isto não significa que outras combinações vão falhar imediatamente, mas podem acontecer problemas sutis, e nem tão sutis. Os comandos de contrabarra são os mais propensos a não serem bem-sucedidos se o servidor for de uma versão diferente.

## Notas para os usuários do Windows

O psql é construído como um “aplicativo console”. Como as janelas console do Windows utilizam uma página de código diferente do restante do sistema, deve-se tomar um cuidado especial quando são utilizados caracteres de 8 bits no psql. Quando o psql detecta uma página de código da console problemática, emite uma advertência na inicialização. Para mudar a página de código da console são necessárias duas coisas:

- Definir a página de código executando `\!chcp 1252` ( `1252` (<http://www.microsoft.com/globaldev/reference/sbcs/1252.htm>) é a página de código apropriada para o Português do Brasil; se for necessário deve ser utilizado outro código de página). Se estiver sendo utilizado o Cygwin, este comando pode ser colocado em `/etc/profile`.
- Definir a fonte da console como “Lucida Console”, porque a fonte mapa de bits ( raster ([http://en.wikipedia.org/wiki/Bitmap\\_font](http://en.wikipedia.org/wiki/Bitmap_font)) ) não trabalha com páginas de código ANSI.

## Exemplos

O primeiro exemplo mostra como distribuir um comando por várias linhas de entrada. Deve ser observada a mudança do prompt:

```
testdb=> CREATE TABLE minha_tabela (
testdb(>     primeiro INTEGER NOT NULL DEFAULT 0,
testdb(>     segundo TEXT
testdb-> );
CREATE TABLE
```

Abaixo está mostrada a definição da tabela:

```
testdb=> \d minha_tabela
      Tabela "public.minha_tabela"
      Coluna | Tipo | Modificadores
-----+-----+-----
primeiro | integer | not null default 0
segundo | text |
```

Agora o prompt será mudado para algo mais interessante:

```
testdb=> \set PROMPT1 '%n%m %~%R%# '
peter@localhost testdb=>
```

Assumindo que a tabela já esteja com dados e queremos vê-los:

```
peter@localhost testdb=> SELECT * FROM minha_tabela;
primeiro | segundo
-----+-----
1 | UM
2 | DOIS
3 | TRÊS
4 | QUATRO
(4 linhas)
```

As tabelas podem ser mostradas de forma diferente usando o comando \pset:

```
peter@localhost testdb=> \pset border 2
O estilo da borda é 2.
peter@localhost testdb=> SELECT * FROM minha_tabela;
+-----+-----+
| primeiro | segundo |
+-----+-----+
| 1 | UM |
| 2 | DOIS |
| 3 | TRÊS |
| 4 | QUATRO |
+-----+-----+
(4 linhas)
peter@localhost testdb=> \pset border 0
O estilo da borda é 0.
peter@localhost testdb=> SELECT * FROM minha_tabela;
primeiro segundo
-----
1 UM
2 DOIS
3 TRÊS
4 QUATRO
```

```
(4 linhas)
peter@localhost testdb=> \pset border 1
O estilo da borda é 1.
peter@localhost testdb=> \pset format unaligned
O formato de saída é unaligned.
peter@localhost testdb=> \pset fieldsep ','
O separador de campos é ",".
peter@localhost testdb=> \pset tuples_only
Mostrando apenas tuplas.
peter@localhost testdb=> SELECT primeiro, segundo FROM minha_tabela;
1,UM
2,DOIS
3,TRÊS
4,QUATRO
```

Como alternativa, podem ser usados os comandos curtos:

```

peter@localhost testdb=> \a \t \x
O formato de saída é aligned.
Desabilitado mostrar somente tuplas.
Habilitada a exibição expandida.
peter@localhost testdb=> SELECT * FROM minha_tabela;
-[ RECORD 1 ]----
primeiro | 1
segundo  | UM
-[ RECORD 2 ]----
primeiro | 2
segundo  | DOIS
-[ RECORD 3 ]----
primeiro | 3
segundo  | TRÊS
-[ RECORD 4 ]----
primeiro | 4
segundo  | QUATRO

```

**Identificadores SQL em meta-comandos do psql.** Este exemplo mostra a diferença de sintaxe dos identificadores SQL quando usados em comandos SQL e meta-comandos do psql. Não é possível utilizar `foo"BAR"baz` em `CREATE TABLE`, mas é possível utilizar em `\dt`.<sup>11</sup>

```

=> CREATE TABLE "Um nome" "estranho" (foo text);
CREATE TABLE

=> CREATE TABLE foo"BAR"baz (foo text);
ERRO:  erro de sintaxe em ou próximo de "\"BAR\"" no caractere 17

```

```

=> CREATE TABLE "fooBARbaz" (foo text);
CREATE TABLE

```

```

=> \dt

```

```

                Lista de relações
Esquema |      Nome      | Tipo |  Dono
-----+-----+-----+-----
public  | Um nome" estranho | tabela | postgres
public  | fooBARbaz         | tabela | postgres
(2 linhas)

```

```

=> \dt fooBARbaz
Não foi encontrada relação correspondente.

```

```

=> \dt "fooBARbaz"

```

```

                Lista de relações
Esquema |      Nome      | Tipo |  Dono
-----+-----+-----+-----
public  | fooBARbaz      | tabela | postgres
(1 linha)

```

```

=> \dt foo"BAR"baz

```

```

                Lista de relações
Esquema |      Nome      | Tipo |  Dono
-----+-----+-----+-----
public  | fooBARbaz      | tabela | postgres
(1 linha)

```

## Notas

1. Veja no Capítulo 44 um exemplo de como personalizar as mensagens mostradas pelo psql. (N. do T.)
2. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
3. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
4. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
5. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
6. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
7. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
8. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
9. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
10. Exemplo escrito pelo tradutor, não fazendo parte do manual original.
11. Exemplo escrito pelo tradutor, não fazendo parte do manual original.

# vacuumdb

## Nome

vacuumdb — limpa e analisa um banco de dados do PostgreSQL

## Sinopse

```
vacuumdb [opção_de_conexão...] [--full | -f] [--verbose | -v] [--analyze | -z] [--table | -t tabela [(coluna [...]) ] ]  
[nome_do_banco_de_dados]  
vacuumdb [opções_de_conexão...] [--all | -a] [--full | -f] [--verbose | -v] [--analyze | -z]
```

## Descrição

O vacuumdb é um utilitário para limpar um banco de dados do PostgreSQL. O vacuumdb também gera as estatísticas internas usadas pelo otimizador de comandos do PostgreSQL.

O vacuumdb é uma capa em torno do comando *VACUUM* do SQL. Não existe diferença efetiva entre limpar o banco de dados através deste utilitário ou através de outros métodos para acessar o servidor.

## Opções

O vacuumdb aceita os seguintes argumentos de linha de comando:

-a  
--all

Limpa todos bancos de dados.

[-d] *nome\_do\_banco\_de\_dados*  
[--dbname] *nome\_do\_banco\_de\_dados*

Especifica o nome do banco de dados a ser limpo ou analisado. Se não for especificado, e a opção -a (ou --all) não for usada, o nome do banco de dados é obtido a partir da variável de ambiente PGDATABASE. Se esta variável não estiver definida, então é usado o nome do usuário especificado para a conexão.

-e  
--echo

Mostra os comandos que o vacuumdb gera e envia para o servidor.

-f  
--full

Executa a limpeza “completa”.

-q  
--quiet

Não exibe resposta.

-t *tabela* [ (*coluna* [...]) ]  
--table *tabela* [ (*coluna* [...]) ]

Limpa ou analisa somente a *tabela*. Os nomes das colunas só podem ser especificados juntamente com a opção --analyze.

**Dica:** Se forem especificadas as colunas, provavelmente será necessário fazer o escape (\) dos parênteses para o interpretador de linhas de comando (Veja exemplos abaixo).

-v  
--verbose

Mostra informações detalhadas durante o processamento.

`-z`  
`--analyze`

Calcula as estatísticas a serem utilizadas pelo otimizador.

O vacuumdb também aceita os seguintes argumentos de linha de comando para os parâmetros de conexão:

`-h hospedeiro`  
`--host hospedeiro`

Especifica o nome de hospedeiro da máquina onde o servidor está executando. Se o nome iniciar por uma barra (/) é usado como o diretório do soquete do domínio Unix.

`-p porta`  
`--port porta`

Especifica a porta TCP, ou a extensão de arquivo do soquete do domínio Unix local, onde o servidor está atendendo as conexões.

`-U nome_do_usuario`  
`--username nome_do_usuario`

Nome do usuário para conectar.

`-W`  
`--password`

Força a solicitação da senha.

## Ambiente

PGDATABASE  
 PGHOST  
 PGPORT  
 PGUSER

Parâmetros de conexão padrão.

## Diagnósticos

Havendo dificuldade, veja no comando *VACUUM* e no *psql* a explicação dos problemas possíveis e as mensagens de erro. O servidor de banco de dados deve estar executando no hospedeiro de destino. Também se aplicam todas as definições de conexão padrão e as variáveis de ambiente utilizadas pela biblioteca cliente libpq.

## Observações

Pode haver necessidade do vacuumdb se conectar várias vezes ao servidor PostgreSQL, solicitando a senha cada uma destas vezes. Neste caso é conveniente existir o arquivo `$HOME/.pgpass`. Consulte a Seção 27.12 para obter informações adicionais.

## Exemplos

Para limpar o banco de dados teste:

```
$ vacuumdb teste
```

Para limpar e analisar para o otimizador o banco de dados chamado grande\_bd:

```
$ vacuumdb --analyze grande_bd
```

Para limpar uma única tabela chamada foo, no banco de dados chamado xyzzy, e analisar para o otimizador uma única coluna desta tabela chamada bar:

```
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzy
```

## **Consulte também**

*VACUUM*



## III. Aplicativos do servidor PostgreSQL

Esta parte contém informações de referência para os aplicativos e utilitários de suporte do servidor PostgreSQL. Estes comandos só são úteis quando executados no computador onde o servidor de banco de dados está instalado. Outros programas utilitários estão listados em Referência II, *Aplicativos cliente do PostgreSQL*.

# initdb

## Nome

`initdb` — cria um agrupamento de bancos de dados do PostgreSQL

## Sinopse

```
initdb [opção...] --pgdata | -D diretório
```

## Descrição

O utilitário `initdb` cria um agrupamento de bancos de dados do PostgreSQL. Um agrupamento de bancos de dados é uma coleção de bancos de dados gerenciados por uma única instância do servidor.

Criar um agrupamento de banco de dados consiste em criar os diretórios onde os bancos de dados vão residir, gerar as tabelas do catálogo compartilhadas (tabelas que pertencem ao agrupamento como um todo, e não a um determinado banco de dados), e criar o banco de dados `template1`. Quando, mais tarde, for criado um banco de dado, tudo que existe no banco de dados `template1` será copiado. Este banco de dados contém tabelas do catálogo contendo coisas como os tipos de dado nativos.

Embora o `initdb` tente criar o diretório de dados especificado, provavelmente não terá permissão para fazê-lo, porque geralmente o diretório de dados está sob um diretório que pertence ao `root`. Para resolver uma situação como esta deve-se: criar um diretório de dados vazio como `root`; usar o comando `chown` para tornar o usuário do banco de dados o dono deste diretório; executar `su` para se tornar o usuário do banco de dados; finalmente, executar o `initdb` como o usuário do banco de dados.

O `initdb` deve ser executado pelo mesmo usuário que vai executar processo servidor, porque o servidor necessita ter acesso aos arquivos e diretórios criados pelo `initdb`. Como o servidor não deve ser executado pelo `root`, o `initdb` também não deve ser executado pelo `root` (Na verdade, não permite fazê-lo).

O `initdb` inicializa o idioma e a codificação do conjunto de caracteres padrão do agrupamento de banco de dados. A ordem de classificação (`LC_COLLATE`), e as classes de conjunto de caracteres (`LC_CTYPE`, por exemplo, maiúscula, minúscula e dígito), são estabelecidas para toda a existência do agrupamento não podendo ser modificadas. Existe, também, impacto no desempenho quando é escolhida uma ordem de classificação diferente de `C` e de `POSIX`. Por estas razões, é importante fazer a escolha correta ao executar o `initdb`. Outras categorias de idioma podem ser mudadas posteriormente ao iniciar o servidor. Todos os valores de idioma do servidor (`lc_*`) podem ser vistos através do comando `SHOW ALL`. Podem ser encontrados mais detalhes na Seção 20.1.

A codificação do conjunto de caracteres de cada banco de dados pode ser definida individualmente, no momento da criação do banco de dados. O `initdb` determina a codificação do banco de dados `template1`, que serve de padrão para todos os outros bancos de dados. Para alterar a codificação padrão deve ser usada a opção `--encoding`. Podem ser encontrados mais detalhes na Seção 20.2.

## Opções

```
-A método_de_autenticação
```

```
--auth=método_de_autenticação
```

Esta opção especifica o método de autenticação para os usuários locais usado no arquivo de configuração `pg_hba.conf`. Não use `trust` a menos que confie em todos os usuários locais do sistema. `Trust` é o padrão para facilitar a instalação.

```
-D diretório
```

```
--pgdata=diretório
```

Esta opção especifica o diretório onde o agrupamento de banco de dados será armazenado. Esta é a única informação requerida pelo `initdb`, mas pode-se evitar escrevê-la definindo a variável de ambiente `PGDATA`, o que é conveniente

porque, depois, o servidor de banco de dados (`postmaster`) poderá encontrar o diretório do bancos de dados usando esta mesma variável.

```
-E codificação
--encoding=codificação
```

Seleciona a codificação do banco de dados modelo. Também será a codificação padrão para todos os bancos de dados criados posteriormente, a não ser quando for especificada uma outra. O padrão é derivado do idioma, ou `SQL_ASCII` se este não funcionar. Os conjuntos de caracteres suportados pelo servidor PostgreSQL estão descritos na Seção 20.2.1.

```
--locale=idioma
```

Define o idioma padrão para o agrupamento de banco de dados. Se esta opção não for especificada, o idioma é herdado do ambiente onde o `initdb` está executando. O suporte a idioma está descrito na Seção 20.1.

```
--lc-collate=idioma
--lc-ctype=idioma
--lc-messages=idioma
--lc-monetary=idioma
--lc-numeric=idioma
--lc-time=idioma
```

Como `--locale`, mas somente define o idioma para a categoria especificada.

```
-U nome_do_usuario
--username=nome_do_usuario
```

Especifica o nome de usuário do superusuário do banco de dados. Por padrão o nome do usuário executando o `initdb`. Não importa realmente qual seja o nome do superusuário, mas é preferível manter o nome habitual `postgres`, mesmo que o nome do usuário do sistema operacional seja diferente.

```
-W
--pwprompt
```

Faz o `initdb` solicitar a senha a ser atribuída ao superusuário do banco de dados. Se não se pretende utilizar autenticação por senha, isto não tem importância. Senão, não será possível utilizar autenticação por senha enquanto não for atribuída uma senha.

```
--pwfile=nome_do_arquivo
```

Faz o `initdb` ler a senha do superusuário do banco de dados em um arquivo. A primeira linha do arquivo é lida como sendo a senha.

Também estão disponíveis outros parâmetros, menos utilizados:

```
-d
--debug
```

Mostra a saída de depuração do servidor de `bootstrap`, e algumas outras mensagens de menor interesse para o público em geral. O servidor de `bootstrap` é o programa que o `initdb` utiliza para criar as tabelas do catálogo. Esta opção gera uma quantidade imensa de saída extremamente entediante.

```
-L diretório
```

Especifica onde o `initdb` deve encontrar os seus arquivos de entrada para inicializar o agrupamento de banco de dados. Será dito se é necessário especificar o idioma explicitamente.

```
-n
--noclean
```

Por padrão, quando o `initdb` determina que um erro impediu a criação completa do agrupamento de bancos de dados, são removidos todos os arquivos criados antes de ser descoberto que não era possível terminar o trabalho. Esta opção impede a remoção e, portanto, é útil para a depuração.

## **Ambiente**

PGDATA

Especifica o diretório onde o agrupamento de bancos de dados deve ser armazenado; pode ser mudado usando a opção `-D`.

## **Consulte também**

postgres, postmaster

# ipcclean

## Nome

`ipcclean` — remove a memória compartilhada e os semáforos de um servidor PostgreSQL que caiu

## Sinopse

`ipcclean`

## Descrição

O utilitário `ipcclean` remove todos os segmentos de memória compartilhada e os semáforos definidos, pertencentes ao usuário corrente. Sua finalidade é ser usado para fazer a limpeza após a queda do servidor PostgreSQL (`postmaster`). Deve ser observado que reiniciar o servidor imediatamente também limpa a memória compartilhada e os semáforos e, portanto, este utilitário possui pouca utilidade prática.

Somente o administrador do banco de dados deve executar este utilitário, porque pode ocasionar um comportamento bizarro (por exemplo, quedas) se for executado durante uma sessão multiusuária. Se este utilitário for executado enquanto o servidor estiver executando, a memória compartilhada e os semáforos alocados pelo servidor são removidos, podendo ocasionar consequências graves para o servidor.

## Observações

Este script é um “hack”, mas nestes vários anos desde que foi escrito ninguém conseguiu desenvolver uma solução igualmente efetiva e portátil. Como agora o `postmaster` pode se autolimpar, não é provável que o `ipcclean` seja melhorado no futuro.

Este script faz suposições em relação ao formato da saída do utilitário `ipcs`, que podem não ser verdadeiras entre sistemas operacionais diferentes. Portanto, pode ser que não funcione no seu sistema operacional. É aconselhável olhar o script antes de executá-lo.

# pg\_controldata

## Nome

`pg_controldata` — mostra informações de controle de um agrupamento de bancos de dados PostgreSQL

## Sinopse

`pg_controldata` [*diretório\_de\_dados*]

## Descrição

O utilitário `pg_controldata` mostra informações inicializadas durante a execução do `initdb`, tal como a versão do catálogo e o idioma do servidor. Mostra, também, informações sobre a escrita prévia do registro no `log` (WAL) e o processamento dos pontos de controle (`checkpoint`). Estas informações são globais do agrupamento, e não específicas de um determinado banco de dados.

Este utilitário pode ser executado apenas pelo usuário que inicializou o agrupamento, porque necessita de acesso de leitura para o diretório de dados. O diretório de dados pode ser especificado na linha de comando, ou pode ser usada a variável de ambiente `PGDATA`.

## Ambiente

`PGDATA`

Local padrão do diretório de dados.

# pg\_ctl

## Nome

`pg_ctl` — inicia, pára ou reinicia o servidor PostgreSQL

## Sinopse

```
pg_ctl start [-w] [-s] [-D diretório_de_dados] [-l nome_do_arquivo] [-o opções] [-p caminho]  
pg_ctl stop [-W] [-s] [-D diretório_de_dados] [-m s[mart] | f[ast] | i[mmediate] ]  
pg_ctl restart [-w] [-s] [-D diretório_de_dados] [-m s[mart] | f[ast] | i[mmediate] ] [-o opções]  
pg_ctl reload [-s] [-D diretório_de_dados]  
pg_ctl status [-D diretório_de_dados]  
pg_ctl kill [nome_do_sinal] [id_do_processo]
```

## Descrição

O `pg_ctl` é um utilitário para iniciar, parar ou reiniciar o servidor PostgreSQL (postmaster), ou mostrar o status de um servidor ativo. Embora o servidor possa ser iniciado manualmente, o `pg_ctl` encapsula tarefas como redirecionar a saída do `log`, desacoplar do terminal e do grupo de processos de forma adequada, além de fornecer opções convenientes para uma parada controlada.

No modo iniciar (`start`), é lançado um novo servidor. O servidor é iniciado em segundo plano, e a entrada padrão é direcionada para `/dev/null`. A saída padrão e o erro padrão são ambos anexados ao arquivo de `log` (se a opção `-l` for usada), ou redirecionada para a saída padrão do `pg_ctl` (não o erro padrão). Se não for escolhido nenhum arquivo de `log`, a saída padrão do `pg_ctl` deve ser redirecionada para um arquivo ou enviada para outro processo (através de um `pipe`) como, por exemplo, um programa de rotação de `log`, como o `rotatelog`s, senão o `postmaster` escreverá sua saída no terminal que o controla (do segundo plano), e não vai deixar o grupo de processos da `shell`.

No modo parar (`stop`), o servidor que está executando no diretório de dados especificado é parado. Podem ser selecionados três métodos de parada diferentes pela opção `-m`: O modo “Smart” (inteligente) aguarda todos os clientes desconectarem. Este é o padrão. O modo “Fast” (rápido) não aguarda os clientes desconectarem. Todas as transações ativas são desfeitas (`rollback`), os clientes são desconectados à força e, em seguida, o servidor é parado. O modo “Immediate” (imediatamente) interrompe todos os processos servidores sem uma parada limpa, provocando um processamento de recuperação ao reiniciar.

O modo reiniciar (`restart`) executa uma parada seguida por um início. Permite mudar as opções de linha de comando do `postmaster`.

O modo recarregar (`reload`) simplesmente envia o sinal `SIGHUP` para o processo `postmaster`, fazendo este ler novamente os arquivos de configuração (`postgresql.conf`, `pg_hba.conf`, etc.). Permite mudar as opções do arquivo de configuração que não requerem um reinício completo para produzir efeito.

O modo `status` verifica se o servidor está executando no diretório de dados especificado e, se estiver, mostra o PID e as opções de linha de comando usadas para chamá-lo.

O modo `kill` permite enviar um sinal para um determinado processo. É particularmente útil no Microsoft Windows que não possui o comando `kill`. Deve ser utilizado `--help` para ver a lista de nomes de sinais suportados.

## Opções

`-D diretório_de_dados`

Especifica o local dos arquivos de banco de dados no sistema de arquivos. Se for omitido, é usada a variável de ambiente `PGDATA`.

`-l nome_do_arquivo`

Anexa a saída do `log` do servidor ao `nome_do_arquivo`. Se o arquivo não existir é criado. A `umask` é definida como 077 e, portanto, o padrão é não permitir o acesso ao arquivo de `log` pelos outros usuários.

`-m modo`

Especifica o modo de parada (shutdown). O *modo* pode ser `smart`, `fast` ou `immediate`, ou a primeira letra de um desses três.

`-o opções`

Especifica as opções a serem passadas diretamente para o comando `postmaster`.

As opções são geralmente envoltas por apóstrofes (') ou aspas ("), para garantir que sejam passadas como um grupo.

`-p caminho`

Especifica o local do arquivo executável `postmaster`. Por padrão o executável `postmaster` é lido do mesmo diretório do `pg_ctl` ou, caso não seja bem-sucedido, do diretório de instalação especificado. Não é necessário usar esta opção, a menos que esteja sendo feito algo diferente do usual e recebendo mensagem de erro informando que o executável do `postmaster` não foi encontrado.

`-s`

Mostra somente os erros, sem mensagens informativas.

`-w`

Aguarda o início ou a parada terminar. Expira em 60 segundos. Este é o padrão para a parada. Uma parada bem-sucedida é indicada pela remoção do arquivo do PID. Para o início, a execução do comando `psql -l` bem-sucedida indica sucesso. O `pg_ctl` tenta utilizar a porta apropriada para o `psql`. Se a variável de ambiente `PGPORT` existir, esta é usada. Senão, será visto se a porta está definida no arquivo `postgresql.conf`. Se nenhum destes dois for usado, será utilizada a porta padrão com a qual o PostgreSQL foi compilado (5432 por padrão). Quando estiver aguardando, o `pg_ctl` retornará um código de saída acurado baseado no sucesso do início ou da parada.

`-W`

Não aguarda o início ou a parada terminar. Este é o padrão para inícios e reinícios.

## Ambiente

`PGDATA`

Local padrão do diretório de dados.

`PGPORT`

Porta padrão para o `psql` (usada pela opção `-w`).

Para os demais consulte o `postmaster`.

## Arquivos

`postmaster.pid`

A existência deste arquivo no diretório de dados é utilizada para ajudar o `pg_ctl` a determinar se o servidor está executando ou não.

`postmaster.opts.default`

Se existir este arquivo no diretório de dados, o `pg_ctl` (no modo `start`) passa o conteúdo deste arquivo como opções para o comando `postmaster`, a menos que esteja substituído pela opção `-o`.

`postmaster.opts`

Se existir este arquivo no diretório de dados, o `pg_ctl` (no modo `restart`) passa o conteúdo deste arquivo como opções para o comando `postmaster`, a menos que esteja substituído pela opção `-o`. O conteúdo deste arquivo também é mostrado no modo `status`.

`postgresql.conf`

Este arquivo, localizado no diretório de dados, é analisado para descobrir a porta apropriada a ser utilizada pelo `psql` quando a opção `-w` é usada no modo `start`.



## Observações

Aguardar o término do início não é uma operação bem definida, podendo não ser bem-sucedida se o controle de acesso for configurado de modo que o cliente local não possa se conectar sem intervenção manual (por exemplo, autenticação por senha).

## Exemplos

### Iniciar o servidor

Para iniciar o servidor:

```
$ pg_ctl start
```

Exemplo de iniciar o servidor bloqueando até o servidor estar pronto:

```
$ pg_ctl -w start
```

Para um servidor usando a porta 5433, e executando sem `fsync`, deve ser usado:

```
$ pg_ctl -o "-F -p 5433" start
```

### Parar o servidor

```
$ pg_ctl stop
```

pára o servidor. A chave `-m` permite controlar *como* o servidor irá parar.

### Reiniciar o servidor

Reiniciar o servidor praticamente equivale a parar o servidor e iniciá-lo novamente, exceto que o `pg_ctl` salva e reutiliza as opções de linha de comando passadas para a instância executando anteriormente. Para reiniciar o servidor da forma mais simples possível:

```
$ pg_ctl restart
```

Para reiniciar o servidor aguardando o término da parada e da inicialização:

```
$ pg_ctl -w restart
```

Para reiniciar usando a porta 5433 e desabilitando o `fsync` após o reinício:

```
$ pg_ctl -o "-F -p 5433" restart
```

### Mostrar o status do servidor

Abaixo segue um exemplo da saída de status mostrada pelo `pg_ctl`:

```
$ pg_ctl status
```

```
pg_ctl: postmaster is running (pid: 13718)
```

```
Command line was:
```

```
/usr/local/pgsql/bin/postmaster '-D' '/usr/local/pgsql/data' '-p' '5433' '-B' '128'
```

Esta é a linha de comandos que seria usada no modo de reinício.

Abaixo segue um exemplo da saída de status mostrada pelo `pg_ctl` no Windows 2000<sup>1</sup>:

```
C:\Program Files\PostgreSQL\8.0\bin> set PGDATA=C:\Program Files\PostgreSQL\8.0\data
```

```
C:\Program Files\PostgreSQL\8.0\bin> pg_ctl status
```

```
pg_ctl: postmaster is running (PID: 840)
```

```
C:/Program Files/PostgreSQL/8.0/bin/postmaster.exe "-D" "C:/Program Files/PostgreSQL/8.0/data"
```

## Consulte também

postmaster

## **Notas**

1. Exemplo escrito pelo tradutor, não fazendo parte do manual original.

# pg\_resetxlog

## Nome

`pg_resetxlog` — redefine o conteúdo do log de escrita prévia e outras informações de controle de um agrupamento de bancos de dados do PostgreSQL

## Sinopse

```
pg_resetxlog [-f] [-n] [-o oid] [-x xid] [-l timelineid,fileid,seg] diretório_de_dados
```

## Descrição

O utilitário `pg_resetxlog` limpa o log de escrita prévia (WAL) e, opcionalmente, redefine algumas outras informações de controle (armazenadas no arquivo `pg_control`). Algumas vezes esta função é necessária quando estes arquivos ficam corrompidos. Deve ser utilizada apenas como último recurso, quando o servidor não iniciar por causa da corrupção destes arquivos.

Após executar este comando deve ser possível iniciar o servidor, mas deve-se ter em mente que o banco de dados pode conter dados inconsistentes devido à presença de transações parcialmente efetivadas. Deve ser feita, imediatamente, uma cópia de segurança dos dados, executar o `initdb` e recarregar os dados. Após a recarga as inconsistências devem ser verificadas e corrigidas conforme necessário.

Este utilitário pode ser executado apenas pelo usuário que instalou o servidor, porque requer acesso de leitura/gravação no diretório de dados. Por motivo de segurança, o diretório de dados deve ser especificado na linha de comando. O `pg_resetxlog` não utiliza a variável de ambiente `PGDATA`.

Se o `pg_resetxlog` informar que não está conseguindo determinar os dados válidos para `pg_control`, pode ser forçado a prosseguir assim mesmo especificando a chave `-f` (forçar). Neste caso, são utilizados valores plausíveis para os dados que estão faltando. Pode-se esperar que a maioria dos campos correspondam, mas pode ser necessário auxílio manual para os campos próximo OID, próximo ID de transação, endereço inicial do WAL e idioma do banco de dados. Os três primeiros podem ser definidos usando as chaves discutidas abaixo. O próprio ambiente do `pg_resetxlog` é a fonte para os campos de idioma; deve-se cuidar para que `LANG` e as demais variáveis de ambiente análogas correspondam ao ambiente em que o `initdb` foi executado. Se não for possível determinar o valor correto para todos estes campos o `-f` ainda pode ser usado, mas o banco de dados recuperado deve ser tratado como ainda mais suspeito que o usual: uma imediata cópia de segurança e sua recarga é imperativa. Não deve ser executada nenhuma operação que modifique os dados do banco de dados antes de ser feita a cópia de segurança, porque uma atividade deste tipo pode piorar ainda mais a situação.

As chaves `-o`, `-x` e `-l` permitem definir manualmente o próximo OID, o próximo ID de transação e o endereço inicial do WAL. Somente são necessários quando `pg_resetxlog` não for capaz de determinar os valores apropriados por meio da leitura do arquivo `pg_control`. Um valor seguro para o identificador da próxima transação pode ser determinado verificando o nome de arquivo com o maior valor numérico no diretório `/pg_clog` sob o diretório de dados, somando um, e depois multiplicando por 1.048.576. Deve ser observado que os nomes dos arquivos estão em hexadecimal. Normalmente é mais fácil especificar o valor da chave em hexadecimal também. Por exemplo, se 0011 for a maior entrada em `pg_clog`, então `-x 0x1200000` servirá (cinco zeros à direita fornecem o multiplicador apropriado). O endereço inicial do WAL deve ser maior do que qualquer nome de arquivo existente no diretório `/pg_xlog` sob o diretório de dados. Estes nomes também estão em hexadecimal, e possuem três partes. A primeira parte é o “timeline ID” e deve, geralmente, ser mantido o mesmo. Não escolha um valor maior que 255 (0xFF) para a terceira parte; em vez disso incremente a segunda parte e redefine a terceira parte como 0. Por exemplo, se 00000001000000320000004A for a maior entrada em `pg_xlog`, `-l 0x1,0x32,0x4B` servirá; mas se a maior entrada for 000000010000003A000000FF, então deve ser escolhido `-l 0x1,0x3B,0x0` ou maior. Não existe nenhuma forma mais fácil para determinar o próximo OID acima do maior existente no banco de dados, mas por sorte não é crítico definir o próximo OID corretamente.

A chave `-n` (nenhuma operação) faz o `pg_resetxlog` mostrar os valores reconstruídos a partir de `pg_control` e terminar em seguida, sem modificar nada. Isto é principalmente uma ferramenta de depuração, mas pode ser útil para fazer uma verificação antes de permitir que o `pg_resetxlog` realmente efetue a operação.

## **Observações**

Este comando não deve ser usado quando o servidor estiver executando. O `pg_resetxlog` se recusa a iniciar quando encontra o arquivo de bloqueio do servidor no diretório de dados. Se o servidor caiu, então o arquivo de bloqueio pode ter sido deixado no diretório; neste caso, o arquivo de bloqueio deve ser removido para permitir o `pg_resetxlog` executar, mas antes de remover deve-se ter certeza que nem o `postmaster`, nem nenhum processo servidor, ainda está executando.

# postgres

## Nome

`postgres` — executa o servidor PostgreSQL no modo monousuário

## Sinopse

```
postgres [-A 0 | 1 ] [-B número_de_buffers] [-c nome=valor] [-d nível_de_depuração] [--describe-config]
[-D diretório_de_dados] [-e] [-E] [-fs | i | t | n | m | h ] [-F] [-N] [-o nome_do_arquivo] [-O] [-P] [-s | -t pa | pl |
ex ] [-S memória_de_trabalho] [-W segundos] [--nome=valor] nome_do_banco_de_dados
postgres [-A 0 | 1 ] [-B número_de_buffers] [-c nome=valor] [-d nível_de_depuração] [-D
diretório_de_dados] [-e] [-f s | i | t | n | m | h ] [-F] [-o nome_do_arquivo] [-O] [-p
nome_do_banco_de_dados] [-P] [-s | -t pa | pl | ex ] [-S memória_de_trabalho] [-v protocolo] [-W
segundos] [--nome=valor]
```

## Descrição

O executável `postgres` é o verdadeiro processo servidor PostgreSQL que processa os comandos. Normalmente não é chamado diretamente; em vez deste é iniciado o servidor multiusuário `postmaster`.

A segunda forma acima é como o `postgres` é chamado pelo `postmaster` (somente conceitualmente, porque o `postmaster` e o `postgres` são, na verdade, o mesmo programa); não deve ser chamado diretamente desta maneira. A primeira forma chama o servidor diretamente no modo monousuário interativo. A principal utilização deste modo é durante a inicialização pelo `initdb`. Algumas vezes é utilizada para depuração ou para recuperação de desastre.

Quando chamado no modo interativo a partir da linha de comando, o usuário pode entrar com comandos e os resultados são mostrados na tela, mas de uma forma que é mais útil para os desenvolvedores do que para os usuários finais. Deve-se notar que executar o servidor em modo monousuário não é inteiramente adequado para a depuração do servidor, uma vez que não acontecerá nenhuma comunicação entre processos e bloqueio de fato.

Ao executar o servidor autônomo, o usuário da sessão será definido como o usuário com o identificador 1. Não é necessário este usuário existir e, portanto, o servidor autônomo pode ser usado para recuperar manualmente certos tipos de dano acidentais dos catálogos do sistema. No modo autônomo são concedidos poderes implícitos de superusuário ao usuário com identificador igual a 1.

## Opções

Quando o `postgres` é iniciado pelo `postmaster`, herda todas as opções definidas por este. Além disso, opções específicas do `postgres` podem ser passadas pelo `postmaster` usando a chave `-o`.

Pode-se evitar a digitação destas opções definindo um arquivo de configuração. Consulte a Seção 16.4 para obter detalhes. Algumas opções (seguras) também podem ser definidas, por um modo dependente do aplicativo, pelo cliente se conectando. Por exemplo, se a variável de ambiente `PGOPTIONS` estiver definida, então os clientes baseados na `libpq` passam esta cadeia de caracteres para o servidor, que irá interpretá-la como opções de linha de comando do `postgres`.

## Finalidade geral

As opções `-A`, `-B`, `-c`, `-d`, `-D`, `-F` e `--nome` possuem o mesmo significado que no `postmaster`, exceto que `-d 0` impede o nível de log do `postmaster` ser propagado para o `postgres`.

`-e`

Define, para os campos de entrada de data, o estilo padrão da data como “European”, que é a ordem `DMY`. Também faz o dia vir antes do mês em certos formatos de saída de data. Consulte a Seção 8.5 para obter informações adicionais.

`-o nome_do_arquivo`

Envia toda saída do log do servidor para `nome_do_arquivo`. Se o `postgres` estiver executando sob o `postmaster` esta opção será ignorada, e a `stderr` herdada do `postmaster` será utilizada.

-P

Ignora os índices do sistema ao ler as tabelas do sistema (mas continua atualizando os índices ao modificar as tabelas). É útil ao se fazer a recuperação por causa de índices do sistema corrompidos.

-S

Mostra a informação de tempo, e outras estatísticas, ao final de cada comando. Útil para avaliações ou para definir o número de buffers.

-S *memória\_de\_trabalho*

Especifica a quantidade de memória a ser usada pelas ordenações e hashes internos, antes de recorrer a arquivos temporários em disco. Consulte a descrição do parâmetro de configuração `work_mem` na Seção 16.4.3.1.

## Opções para o modo autônomo

*nome\_do\_banco\_de\_dados*

Especifica o nome do banco de dados a ser acessado. Se for omitido o padrão é o nome do usuário.

-E

Mostra todos os comandos.

-N

Desabilita o uso do caractere de nova-linha como delimitador do comando.

## Opções semi-internas

Existem várias outras opções que podem ser especificadas, usadas principalmente para fins de depuração, mostradas apenas para uso pelos desenvolvedores de sistema do PostgreSQL. *A utilização de qualquer uma destas opções é altamente desaconselhada.* Além disso, qualquer uma destas opções poderá mudar ou desaparecer em uma versão futura sem nenhum aviso.

-f { s | i | m | n | h }

Proíbe o uso de um determinado método de varredura ou de junção: `s` e `i` desabilitam a varredura seqüencial e de índice, respectivamente, enquanto `n`, `m` e `h` desabilitam as junções de laço-aninhado, mesclagem e hash, respectivamente.

**Nota:** Nem as varreduras seqüenciais nem as junções de laço aninhado podem ser desabilitadas completamente; as opções `-fs` e `-fn` simplesmente desencorajam o uso pelo otimizador de planos deste tipo, se houver alguma outra alternativa.

-O

Permite modificar a estrutura das tabelas do sistema. Usada pelo `initdb`.

-p *nome\_do\_banco\_de\_dados*

Indica que este processo foi iniciado pelo `postmaster` e especifica o banco de dados a ser utilizado, etc.

-t pa[rser] | pl[anner] | e[xecutor]

Mostra estatísticas de tempo para cada comando, relacionando com cada um dos principais módulos do sistema. Esta opção não pode ser usada junto com a opção `-s`.

-v *protocolo*

Especifica o número da versão do protocolo cliente/servidor a ser usado por esta sessão.

-W *segundos*

Tão logo esta opção seja encontrada, o processo adormece pela quantidade especificada de segundos, dando ao desenvolvedor tempo para anexar o depurador ao processo servidor.

--describe-config

Esta opção mostra as variáveis de configuração internas do servidor, descrições e padrões, em um formato delimitado por tabulação do `COPY`. Foi projetado para ser utilizado principalmente por ferramentas de administração.

## Ambiente

PGDATA

Local padrão do diretório de dados.

Para as outras variáveis de ambiente, com pouca influência durante o modo monousuário, consulte o postmaster.

## Observações

Para cancelar a execução de um comando, deve ser enviado o sinal `SIGINT` para o processo `postgres` executando este comando.

Para fazer o `postgres` recarregar os arquivos de configuração, deve ser enviado o sinal `SIGHUP`. Normalmente é melhor enviar o sinal `SIGHUP` para o `postmaster` que, por sua vez, enviará o sinal `SIGHUP` para cada um de seus descendentes. Mas em alguns casos pode-se querer que apenas um processo `postgres` recarregue os arquivos de configuração.

O `postmaster` utiliza `SIGTERM` para comunicar a um processo `postgres` que termine normalmente, e `SIGQUIT` para que termine sem a limpeza normal. Estes sinais *não devem* ser utilizados pelos usuários. Também não é aconselhável enviar um `SIGKILL` para um processo `postgres` — o `postmaster` vai interpretar como sendo uma queda do `postgres`, e vai obrigar todos os processos `postgres` descendentes terminarem como parte de seu procedimento padrão de recuperação de quedas.

## Utilização

Inicie o servidor autônomo com um comando do tipo:

```
postgres -D /usr/local/pgsql/data outras_opções meu_banco_de_dados
```

Forneça o caminho correto para o diretório do banco de dados com `-D`, ou garanta que a variável de ambiente `PGDATA` esteja definida. Também especifique o nome do banco de dados em que deseja trabalhar.

Normalmente, o servidor autônomo trata o caractere de nova-linha como fim da entrada do comando; não existe inteligência sobre ponto-e-vírgula, como existe no `psql`. Para continuar o comando por várias linhas, deve ser digitado uma contrabarra (`\`) logo antes de cada nova-linha, exceto a última.

Mas se for usada a chave de linha de comando `-N`, o caractere de nova-linha não termina a entrada do comando. Neste caso, o servidor lê a entrada padrão até a marca de fim-de-arquivo (EOF) e, então, processa a entrada como sendo a cadeia de caracteres de um único comando. A seqüência contrabarra nova-linha não recebe tratamento especial neste caso.

Para sair da sessão tecle EOF (**Control+D**, geralmente). Se for utilizado `-N`, serão necessários dois EOF consecutivos para sair.

Deve ser observado que o servidor autônomo não oferece funcionalidades sofisticadas para edição de linha (por exemplo, não existe o histórico dos comandos).

## Consulte também

initdb, ipcclean, postmaster

# postmaster

## Nome

`postmaster` — servidor de banco de dados multiusuário do PostgreSQL

## Sinopse

```
postmaster [-A 0 | 1 ] [-B número_de_buffers] [-c nome=valor] [-d nível_de_depuração] [-D diretório_de_dados] [-F] [-h nome_do_hospedeiro] [-i] [-k diretório] [-l] [-N número_máximo_de_conexões] [-o opções_extras] [-p porta] [-S] [--nome=valor] [-n | -s]
```

## Descrição

O `postmaster` é o servidor de banco de dados multiusuário PostgreSQL. Para um aplicativo cliente acessar um banco de dados deve se conectar (através de uma rede ou localmente) ao `postmaster`. O `postmaster`, então, inicia um processo servidor separado (“`postgres`”) para tratar a conexão. O `postmaster` também gerencia a comunicação entre os processos servidor.

Por padrão, o `postmaster` inicia em primeiro plano (`foreground`) e envia as mensagens de `log` para a saída de erro padrão. Para uso prático o `postmaster` deve ser iniciado como um processo em segundo plano (`background`), provavelmente durante a inicialização do sistema operacional.

Um `postmaster` gerencia sempre os dados de, precisamente, um agrupamento de bancos de dados. Um agrupamento de bancos de dados é uma coleção de bancos de dados armazenados em um local comum no sistema de arquivos (a “área de dados”). Mais de um processo `postmaster` pode estar executando no sistema operacional ao mesmo tempo, desde que utilizem áreas de dados diferentes e portas de comunicação diferentes (veja abaixo). A área de dados é criada pelo `initdb`.

Ao iniciar, o `postmaster` precisa conhecer o local onde está a área de dados. O local deve ser especificado pela opção `-D`, ou por meio da variável de ambiente `PGDATA`; não existe nenhum valor padrão. Normalmente `-D` ou `PGDATA` apontam diretamente para o diretório da área de dados criada pelo `initdb`. Outras disposições de arquivo possíveis estão discutidas na Seção 16.4.1.

## Opções

O `postmaster` aceita os argumentos de linha de comando mostrados abaixo. Para uma discussão detalhada destas opções consulte a Seção 16.4. É possível evitar a digitação da maior parte destas opções usando o arquivo de configuração.

`-A 0 | 1`

Habilita a verificação das asserções <sup>1</sup> em tempo de execução, o que é uma ajuda de depuração para detectar erros de programação. Só está disponível quando habilitada durante a compilação do PostgreSQL. Se for, o padrão é ativa.

`-B número_de_buffers`

Define o número de `buffers` compartilhados para uso pelos processos servidor. O valor padrão deste parâmetro é escolhido automaticamente pelo `initdb`; consulte a Seção 16.4.3.1 para obter informações adicionais.

`-c nome=valor`

Define o parâmetro em tempo de execução designado. Os parâmetros de configuração suportados pelo PostgreSQL estão descritos na Seção 16.4. A maior parte das outras opções de linha de comando são, na verdade, formas curtas de atribuição destes parâmetros. A opção `-c` pode aparecer várias vezes para definir vários parâmetros.

`-d nível_de_depuração`

Define o nível de depuração. Quanto mais alto for definido este valor, mais saída de depuração será escrita no `log` do servidor. Os valores vão de 1 a 5.

`-D diretório_de_dados`

Especifica o local do diretório de dados e dos arquivos de configuração no sistema de arquivos. Consulte a Seção 16.4.1 para obter detalhes.



-F

Desabilita as chamadas a `fsync` para melhorar o desempenho, correndo o risco de corrupção dos dados na ocorrência de uma falha do sistema. Especificar esta opção equivale a desabilitar o parâmetro de configuração `fsync`. Leia com atenção a documentação antes de usar esta opção!

`--fsync=true` produz o efeito oposto desta opção.

-h *nome\_do\_hospedeiro*

Especifica o nome de hospedeiro ou endereço de IP no qual o `postmaster` atende as conexões dos aplicativos cliente. O valor também pode ser uma lista de endereços separados por espaço, ou `*` para especificar que devem ser ouvidas todas as interfaces disponíveis. Um valor vazio especifica não atender nenhum endereço de IP e, neste caso, somente podem ser usados soquetes do domínio Unix para se conectar ao `postmaster`. Por padrão atende somente o localhost. Especificar esta opção equivale a definir o parâmetro de configuração `listen_addresses`.

-i

Permite os clientes remotos se conectarem via TCP/IP (Domínio da Internet). Sem esta opção somente são aceitas as conexões locais. Esta opção equivale a definir `listen_addresses` como `*` no arquivo de configuração `postgresql.conf`, ou usar `-h`.

Esta opção está obsoleta, uma vez que não permite acesso a todas as funcionalidades de `listen_addresses`. Geralmente é melhor definir `listen_addresses` diretamente.

-k *diretório*

Especifica o diretório do soquete do domínio Unix onde o `postmaster` está atendendo as conexões dos aplicativos cliente. Normalmente o padrão é `/tmp`, mas pode ser mudado na compilação.

-l

Habilita as conexões seguras usando SSL. O PostgreSQL deve ter sido compilado com suporte a SSL para ser possível o uso desta opção. Para obter informações adicionais sobre o uso do SSL, consulte a Seção 16.7.

-N *número\_máximo\_de\_conexões*

Define o número máximo de conexões de clientes aceitas por este `postmaster`. Por padrão este valor é 32, mas pode ser definido tão alto quanto o sistema operacional suportar (Deve ser observado que o valor da opção `-B` deve ser pelo menos o dobro do valor da opção `-N`. Veja na Seção 16.5 a discussão sobre os requisitos de recursos do sistema necessários para a conexão de um grande número de clientes). Especificar esta opção equivale a definir o parâmetro de configuração `max_connections`.

-o *opções\_extras*

As opções no estilo linha de comando especificadas nas *opções\_extras* são passadas para todos os processos servidor iniciados por este `postmaster`. Consulte o `postgres` para ver as possibilidades. Se a cadeia de caracteres contendo a opção contiver espaços, toda a cadeia de caracteres deve vir entre apóstrofes (`'`).

-p *porta*

Especifica a porta TCP/IP, ou a extensão do arquivo de soquete do domínio Unix local, onde o `postmaster` está atendendo as conexões dos aplicativos cliente. O padrão é obter o valor a partir da variável de ambiente `PGPORT`, se esta estiver definida, senão usar o valor padrão compilado (normalmente 5432). Se for especificada uma porta diferente da porta padrão, então todos os aplicativos cliente devem especificar a mesma porta usando a opção de linha de comando ou a variável de ambiente `PGPORT`.

-S

Especifica que o processo `postmaster` deve iniciar no modo silencioso, ou seja, será dissociado do terminal (controlador) do usuário, iniciará seu próprio grupo de processos e redirecionará sua saída padrão e erro padrão para `/dev/null`.

O uso desta chave descarta toda a saída para o `log`, o que provavelmente não é o desejado, porque torna muito difícil a solução dos problemas. Veja abaixo uma maneira melhor de iniciar o `postmaster` em segundo plano.

`--silent-mode=false` produz o efeito oposto desta opção.

`--nome=valor`

Define o parâmetro em tempo de execução designado; uma forma mais curta da opção `-c`.

Estão disponíveis duas opções de linha de comando adicionais para a depuração dos problemas que fazem o servidor terminar anormalmente. A estratégia comum nesta situação é notificar todos os outros processos servidor que estes devem terminar e, em seguida, reinicializar a memória compartilhada e os semáforos. Isto é porque o processo servidor com problema pode ter corrompido algum estado compartilhado antes de terminar. Estas opções selecionam comportamentos alternativos do `postmaster` nesta situação. *Nenhuma destas opções foi feita para ser usada durante a operação normal.*

As opções caso-especial são:

`-n`

O `postmaster` não reinicializa as estruturas de dado compartilhadas. Um programador de sistemas com conhecimento adequado poderá, então, usar um depurador para examinar a memória compartilhada e o estado do semáforo.

`-s`

O `postmaster` pára todos os outros processos servidor enviando o sinal `SIGSTOP`, mas não faz com que terminem, permitindo os programadores de sistema coletar “core dumps” de todos os processos servidor manualmente.

## Ambiente

`PGCLIENTENCODING`

A codificação de caracteres padrão utilizada pelos clientes (Os clientes podem substituí-la individualmente). Este valor também pode ser definido no arquivo de configuração.

`PGDATA`

Local padrão do diretório de dados.

`PGDATESTYLE`

Valor padrão do parâmetro em tempo de execução `DateStyle` (A utilização desta variável de ambiente está obsoleta).

`PGPORT`

Porta padrão (definida de preferência no arquivo de configuração).

`TZ`

Zona horária do servidor.

## Diagnósticos

Uma mensagem de erro mencionando `semget` ou `shmget` provavelmente indica a necessidade de configurar o núcleo (`kernel`) para fornecer uma quantidade de memória compartilhada e semáforos adequados. Para obter mais informações consulte a Seção 16.5.

**Dica:** Pode ser possível adiar a reconfiguração do núcleo diminuindo `shared_buffers`, para reduzir o consumo de memória compartilhada do PostgreSQL, e/ou reduzindo `max_connections`, para reduzir o consumo de semáforos.

Uma mensagem de erro sugerindo que um outro `postmaster` está executando deve ser verificada cuidadosamente utilizando, por exemplo, o comando

```
$ ps ax | grep postmaster
```

ou

```
$ ps -ef | grep postmaster
```

dependendo do sistema operacional. Havendo certeza de que não há outro `postmaster` conflitante executando, deve ser removido o arquivo de bloqueio mencionado na mensagem e tentado novamente.

Uma mensagem de erro indicando não ser possível vincular a porta, pode indicar que a porta já esteja em uso por um processo não-PostgreSQL. Este erro também pode acontecer se o `postmaster` for terminado e reiniciado imediatamente usando a mesma porta; neste caso, se deve simplesmente aguardar uns poucos segundos o sistema operacional fechar a porta antes de tentar novamente. Por fim, este erro pode acontecer se for especificado um número de porta que o sistema operacional considere reservado. Por exemplo, muitas versões do Unix consideram os números de porta abaixo de 1024 como “trusted” (confiável) só permitindo o acesso aos superusuários do Unix.

## Observações

Sempre que for possível *não* deve ser usado `SIGKILL` para terminar o `postmaster`. Este sinal impede que o `postmaster` libere os recursos do sistema utilizados (por exemplo, memória compartilhada e semáforos) antes de terminar, podendo causar problemas ao iniciar uma nova execução do `postmaster`.

Para terminar o `postmaster` normalmente, podem ser usados os sinais `SIGTERM`, `SIGINT` e `SIGQUIT`. O primeiro aguarda todos os clientes terminarem antes de fechar, o segundo força a desconexão de todos os clientes e o terceiro fecha imediatamente sem um shutdown adequado, provocando a execução da recuperação ao reiniciar. O sinal `SIGHUP` recarrega os arquivos de configuração do servidor.

O utilitário `pg_ctl` pode ser usado para iniciar e terminar o `postmaster` com segurança e conforto.

As opções `--` não funcionam no FreeBSD nem no OpenBSD. Use o `-c` em seu lugar. Esta é uma falha destes sistemas operacionais; uma versão futura do PostgreSQL disponibilizará um recurso para contornar este problema, caso não seja corrigido.

## Exemplos

Para iniciar o `postmaster` em segundo plano usando os valores padrão:

```
$ nohup postmaster >logfile 2>&1 </dev/null &
```

Para iniciar o `postmaster` usando uma porta específica:

```
$ postmaster -p 1234
```

Este comando inicia o `postmaster` se comunicando através da porta 1234. Para se conectar a este `postmaster` usando o `psql`, deve-se executar:

```
$ psql -p 1234
```

ou definir a variável de ambiente `PGPORT`:

```
$ export PGPORT=1234
$ psql
```

Parâmetros em tempo de execução identificados pelo nome podem ser definidos usando um destes estilos:

```
$ postmaster -c work_mem=1234
$ postmaster --work-mem=1234
```

As duas formas substituem o que estiver definido para `work_mem` no arquivo de configuração `postgresql.conf`. Deve ser observado que os sublinhados nos nomes dos parâmetros podem ser escritos na linha de comando com o caractere sublinhado ou hífen.

**Dica:** Exceto para experimentos de curta duração, provavelmente é uma prática melhor editar as definições no arquivo `postgresql.conf` do que depender das chaves de linha de comando para definir os parâmetros.

## Consulte também

`initdb`, `pg_ctl`

## Notas

1. asserção — do Lat. *assertione* — proposição que se apresenta como verdadeira. PRIBERAM - Língua Portuguesa On-Line (<http://www.priberam.pt/dlpo/dlpo.aspx>). (N. do T.)

## VII. Internamente

Esta parte contém diversas informações úteis para os desenvolvedores do PostgreSQL.

## Capítulo 40. Visão geral da estrutura interna do PostgreSQL

**Autor:** Este capítulo se originou como parte da Tese de Mestrado de Stefan Simkovics, preparada na Universidade de Tecnologia de Viena, sob a direção de O.Univ.Prof.Dr. Georg Gottlob e Univ.Ass.Mag. Katrin Seyr, *Enhancement of the ANSI SQL Implementation of PostgreSQL*.

Este capítulo proporciona uma visão geral da estrutura interna do servidor PostgreSQL. Após a leitura das seções que vêm a seguir, deve-se ter uma idéia de como os comandos são processados. Este capítulo não tem por objetivo fornecer uma descrição detalhada da operação interna do PostgreSQL, uma vez que um documento deste tipo seria muito extenso. Em vez disso, este capítulo tem por objetivo ajudar o leitor entender a seqüência geral das operações que ocorrem dentro do servidor, do ponto em que o comando é recebido ao ponto em que os resultados são retornados para o cliente.

### 40.1. O caminho do comando

Nesta seção é dada uma visão geral resumida dos estágios pelos quais o comando tem que passar para chegar ao resultado.

1. Deve ser estabelecida uma conexão entre o programa aplicativo e o servidor PostgreSQL. O programa aplicativo transmite um comando para o servidor, e aguarda para receber de volta os resultados transmitidos pelo servidor.
2. O *estágio de análise* verifica o comando transmitido pelo programa aplicativo com relação à correção da sintaxe, e cria a *árvore de comando*.
3. O *sistema de reescrita* recebe a árvore de comando criada pelo estágio de análise, e procura por alguma *regra* (armazenada nos *catálogos do sistema*) a ser aplicada na árvore de comando. Realiza as transformações especificadas no *corpo das regras*.

Uma das aplicações do sistema de reescrita é a criação de *visões*. Sempre que é executado um comando em uma visão (ou seja, *uma tabela virtual*), o sistema de reescrita reescreve o comando do usuário como um comando acessando as *tabelas base* especificadas na *definição da visão*, em vez da visão.

4. O *planejador/otimizador* recebe a árvore de comando (reescrita), e cria o *plano de comando* que será a entrada do *executor*.

Isto é feito criando primeiro todos os *caminhos* possíveis que levam ao mesmo resultado. Por exemplo, se existe um índice em uma relação a ser varrido, existem dois caminhos para a varredura. Uma possibilidade é uma varredura seqüencial simples, e a outra possibilidade é utilizar o índice. Em seguida é estimado o custo de execução de cada um dos caminhos, e escolhido o mais barato. O caminho mais barato é expandido em um plano completo para que o executor possa utilizá-lo.

5. O executor caminha recursivamente através da *árvore do plano*, e traz as linhas no caminho representado pelo plano. O executor faz uso do *sistema de armazenamento* ao varrer as relações, realiza *classificações* e *junções*, avalia as *qualificações* e, por fim, envia de volta as linhas derivadas.

Nas seções seguintes cada um dos itens listados acima é coberto de forma mais detalhada, para dar uma compreensão melhor do controle interno e das estruturas de dado do PostgreSQL.

### 40.2. Como as conexões são estabelecidas

O PostgreSQL é implementado utilizando um modelo cliente/servidor simples de “um processo por usuário”. Neste modelo existe um *processo cliente* conectado a exatamente um *processo servidor*. Como não se sabe adiantadamente quantas conexões serão realizadas, é utilizado um *processo mestre* que cria um novo processo servidor cada vez que uma conexão é requisitada. Este processo mestre se chama *postmaster*, e fica atendendo a chegada de novas conexões na porta TCP/IP especificada. Sempre que é detectada a requisição de uma nova conexão, o processo *postmaster* cria um novo processo servidor chamado *postgres*. Os servidores tarefa (processos *postgres*) se comunicam entre si utilizando *semáforos* e *memória compartilhada*, para garantir a integridade dos dados nos acessos simultâneos aos dados.

O processo cliente pode ser qualquer programa que compreenda o protocolo do PostgreSQL, descrito no Capítulo 42. Muitos clientes são baseados na biblioteca *libpq* da linguagem C, mas existem várias implementações independentes do protocolo, como o *driver* de JDBC da linguagem Java.

Uma vez estabelecida a conexão, o processo cliente pode enviar comandos para o *servidor* (backend). O comando é transmitido utilizando texto puro, ou seja, não existe análise feita no *cliente* (frontend). O servidor analisa o comando, cria o *plano de execução*, executa o plano, e retorna as linhas obtidas para o cliente transmitindo-as através da conexão estabelecida.

### 40.3. O estágio de análise

O *estágio de análise* consiste de duas partes:

- O *analisador* definido em `gram.y` e `scan.l` é construído utilizando as ferramentas do Unix yacc e lex.
- O *processo de transformação* faz modificações e ampliações nas estruturas de dados retornadas pelo analisador.

#### 40.3.1. O analisador

O analisador (*parser*) precisa verificar a validade da cadeia de caracteres (que chega como texto ASCII puro) com relação à sintaxe.<sup>1</sup> Se a sintaxe estiver correta, é construída uma *árvore de análise* e enviada de volta; senão, é retornada uma condição de erro. O analisador e o analisador léxico<sup>2</sup> (*lexer*) são implementados utilizando as ferramentas bem conhecidas do Unix yacc e lex.

O *analisador léxico* é definido no arquivo `scan.l`, sendo responsável pelo reconhecimento dos *identificadores*, das *palavras chave do SQL* etc. Para toda palavra chave ou identificador encontrado é gerado um *termo* (*token*), e enviado para o analisador.

O analisador é definido no arquivo `gram.y`, e consiste de um conjunto de *regras gramaticais*<sup>3</sup> e *ações* executadas sempre que uma regra é disparada. O código das ações (que na verdade é um código C) é utilizado para construir a árvore de análise.

O arquivo `scan.l` é transformado no arquivo de código fonte C `scan.c` utilizando o programa `lex`, e `gram.y` é transformado em `gram.c` utilizando o yacc. Após estas transformações serem feitas, pode ser utilizado um compilador C normal para criar o analisador. Não devem ser feitas modificações nos arquivos C gerados, uma vez que estes são sobrescritos quando se executa o `lex` ou o `yacc`.

**Nota:** Normalmente as transformações e compilações mencionadas são realizadas automaticamente utilizando os arquivos *Makefile* presentes na distribuição do código fonte do PostgreSQL.

Uma descrição detalhada do yacc, ou das regras gramaticais contidas em `gram.y`, estão acima do escopo desta documentação. Existem vários livros e documentos que tratam do `lex` e do `yacc`. Deve-se estar familiarizado com o yacc antes de começar a estudar a gramática contida no arquivo `gram.y`, senão vai ser impossível entender o conteúdo deste arquivos.

#### 40.3.2. O processo de transformação

O estágio de análise cria uma árvore de análise utilizando somente regras fixadas sobre a estrutura sintática do SQL. Não faz qualquer procura nos catálogos do sistema, portanto não tem possibilidade de compreender os detalhes da semântica<sup>4</sup> das operações requisitadas. Após o término da análise, o *processo de transformação* recebe a árvore retornada pelo analisador como entrada, e faz a interpretação semântica necessária para compreender quais tabelas, funções e operadores são referenciados pelo comando. A estrutura de dados construída para representar esta informação é chamada de *árvore de comando*.

O motivo para separar a análise intacta (*raw*) da análise semântica é que a procura nos catálogos do sistema só pode ser feita dentro de uma transação, e não se deseja iniciar uma transação imediatamente após receber a cadeia de caracteres do comando. O estágio de análise intacta é suficiente para identificar os comandos de controle de transação (`BEGIN`, `ROLLBACK`, etc.), e os que podem ser executados corretamente sem mais análise. Uma vez descoberto que está se lidando com um comando verdadeiro (como `SELECT` ou `UPDATE`), é correto iniciar a transação caso já não se esteja em uma. Somente então o processo de transformação pode ser chamado.

A árvore de comando criada pelo processo de transformação é estruturalmente semelhante à árvore de análise intacta na maioria dos lugares, mas possui muitas diferenças nos detalhes. Por exemplo, um nodo `FuncCall` na árvore de análise representa algo que se parece sintaticamente com uma chamada de função. Pode ser transformado em um nodo `FuncExpr` ou `Aggref`, dependendo do nome referenciado ser uma função comum ou uma função de agregação. Também são

adicionadas à árvore de comando as informações sobre o verdadeiro tipo de dado das colunas e dos resultados das expressões.

## 40.4. O sistema de regras do PostgreSQL

O PostgreSQL dá suporte a um poderoso *sistema de regras* para a especificação de *visões* e *atualizações de visões* ambíguas. Originalmente o sistema de regras do PostgreSQL consistia de duas implementações:

- A primeira implementação trabalhava utilizando o processamento no *nível de linha*, e era implementada no *executor*. O sistema de regras era chamado sempre que uma linha era acessada. Esta implementação foi removida em 1995 quando a última versão oficial do projeto de Berkeley Postgres foi transformada no Postgres95.
- A segunda implementação do sistema de regras era uma técnica chamada de *reescrita de comando*. O *sistema de reescrita* é um módulo que fica entre o *estágio de análise* e o *planejador/otimizador*. Esta técnica ainda é implementada.

O reescritor de regras está discutido em algum detalhe no Capítulo 33, portanto não é necessário discuti-lo novamente neste capítulo. Somente será destacado que tanto a entrada quanto a saída do reescritor são árvores de comando, ou seja, não existe alteração na representação ou no nível de detalhamento semântico nas árvores. A reescrita pode ser vista como uma expansão de macro.

## 40.5. Planejador/Otimizador

A tarefa do *planejador/otimizador* é criar um plano de execução ótimo. Um dado comando SQL (e, portanto, uma árvore de comando) pode, na verdade, ser executada de várias maneiras diferentes, cada uma das quais produzindo o mesmo conjunto de resultados. Se for computacionalmente praticável, o otimizador de comandos examina cada um dos planos de execução possíveis para, no fim, selecionar o plano de execução que espera ser o mais rápido.

**Nota:** Em algumas situações, o exame de todas as formas pelas quais um comando pode ser executado leva a um consumo excessivo de tempo e de espaço em memória. Em particular, estas situações ocorrem quando se executa comandos que envolvem um grande número de operações de junção. Para ser possível determinar um plano de comando razoável (não o ótimo), em um espaço de tempo razoável, o PostgreSQL utiliza o *Genetic Query Optimizer*.

Na verdade, o procedimento de procura do planejador trabalha com estruturas de dados chamadas de *caminhos* (*paths*), que são simplesmente representações reduzidas dos planos, contendo somente as informações necessárias para o planejador tomar suas decisões. Após ser determinado o caminho mais barato, é construída a *árvore de plano* pronta para ser passada para o executor. Esta árvore representa o plano de execução desejado no nível de detalhamento suficiente para o executor processá-la. No restante desta seção será ignorada a distinção entre caminhos e planos.

### 40.5.1. Geração dos planos possíveis

O planejador/otimizador inicia gerando planos para varrer individualmente cada relação (tabela) utilizada no comando. Os planos possíveis são determinados pelos índices disponíveis em cada relação. Sempre existe a possibilidade de realizar a varredura sequencial da relação, portanto o plano para varredura sequencial é sempre criado. Assumindo que haja um índice definido em uma relação (por exemplo um índice árvore-B), e o comando contenha a restrição `relação.atributo OPR constante`, se acontecer de `relação.atributo` corresponder à chave do índice árvore-B, e `OPR` for um dos operadores listados na *classe de operadores* do índice, é criado um outro plano utilizando o índice árvore-B para varrer a relação. Se existirem outros índices presentes, e a restrição no comando corresponder à chave do índice, serão levados em consideração outros planos.

Após terem sido encontrados todos os planos viáveis para varrer uma única relação, são criados planos para juntar as relações. O planejador/otimizador considera preferencialmente junções entre quaisquer duas relações para as quais existe uma cláusula de junção correspondente na qualificação do `WHERE` (ou seja, para as quais existe uma restrição do tipo `WHERE rel1.atrib1=rel2.atrib2`). Os pares de junção sem cláusula de junção são considerados somente quando não há outra escolha, ou seja, uma determinada relação não tem disponível cláusula de junção com qualquer outra relação. São gerados todos os planos possíveis para cada par de junção considerado pelo planejador/otimizador. As três estratégias possíveis são:

- *junção de laço aninhado*: A relação da direita é varrida uma vez para cada linha encontrada na relação da esquerda. Esta estratégia é fácil de ser implementada, mas pode consumir muito tempo (Entretanto, se a relação da direita puder ser varrida através de uma varredura de índice, esta é uma boa estratégia. É possível utilizar valores da linha corrente da relação da esquerda como chaves para a varredura de índice da relação da direita).

- *junção por classificação e mesclagem*: Cada relação é classificada pelo atributo de junção antes do início da junção. As duas relações são varridas em paralelo, e as linhas correspondentes são combinadas para formar as linhas juntadas. Este tipo de junção é mais atrativo, porque só é necessário varrer cada relação uma vez. A classificação requerida pode ser obtida por um passo explícito de classificação, ou varrendo a relação na ordem apropriada utilizando um índice na chave de junção.
- *junção hash*: Primeiro, a relação da direita é varrida e carregada numa tabela de hash utilizando os atributos de junção como chaves de hash. Em seguida, a relação da esquerda é varrida e os valores apropriados de cada linha encontrada são utilizados como chave de hash para localizar as linhas correspondentes na tabela.

Quando o comando envolve mais de duas relações, o resultado final deve ser construído através de uma árvore de passos de junção, cada um com duas entradas. O planejador examina as diferentes possibilidades de sequência de junção para descobrir a mais barata.

A árvore do plano pronta consiste de varreduras sequenciais ou de índice das relações base, mais nodos de junção de laço aninhado, mesclagem e hash conforme necessário, mais os passos auxiliares necessários, como nodos de classificação ou nodos de cálculo de funções de agregação. A maioria destes tipos de nodos de plano possuem a capacidade adicional de realizar *seleção* (desprezar as linhas que não correspondem à condição booleana especificada) e *projeção* (cálculo de um conjunto de colunas derivadas baseado em valores de coluna fornecidos, ou seja, avaliação de expressões escalares onde for necessário). Uma das responsabilidades do planejador é anexar as condições de seleção da cláusula *WHERE* e os cálculos das expressões de saída requeridos ao nodo mais apropriado da árvore do plano.

## 40.6. Executor

O *executor* recebe o plano retornado pelo planejador/otimizador, e o processa recursivamente para extrair o conjunto de linhas requisitadas. É essencialmente um mecanismo de canal de envio de informação sob demanda. Toda vez que o nodo do plano é chamado deve enviar mais uma linha, ou relatar que não há mais linha a ser enviada.

Para fornecer um exemplo concreto, será assumido que o nodo do topo é um nodo *MergeJoin* (junção por mesclagem). Antes da mesclagem poder ser feita, devem ser trazidas duas linhas (uma de cada subplano). Assim, o executor chama a si próprio recursivamente para processar os subplanos (começa pelo subplano anexado à *lefttree* — árvore da esquerda). Digamos que o novo nodo do topo (o nodo do topo do subplano da esquerda) seja um nodo *Sort* (classificação), e novamente haja necessidade de recursão para obter a linha de entrada. O nodo filho do *Sort* deve ser um nodo *SeqScan*, representando a leitura real da tabela. A execução deste nodo faz o executor trazer uma linha da tabela e retorná-la para o nodo que chamou. O nodo *Sort* chama repetitivamente seu nodo filho para obter todas as linhas a serem classificadas. Quando a entrada é exaurida (conforme indicado pelo nodo filho retornando nulo em vez de uma linha), o código do *Sort* realiza a classificação, e finalmente é capaz de retornar sua primeira linha de saída, que é a primeira linha na ordem da classificação. As demais linhas são mantidas armazenadas, para que possa enviá-las na ordem da classificação em resposta aos próximos comandos.

De maneira semelhante, o nodo *MergeJoin* solicita a primeira linha de seu subplano da direita. Depois compara as duas linhas para verificar se podem ser juntadas; se puderem ser juntadas, retorna a linha juntada para quem chamou. Na próxima chamada, ou imediatamente se não puder juntar as duas entradas, avança para a próxima linha de uma tabela ou da outra (dependendo do que ocorrer na comparação), e novamente compara. No final, um dos dois subplanos ficará exaurido, e o nodo de *MergeJoin* retorna nulo para indicar que não podem ser formadas mais linhas pela junção.

Os comandos complexos podem envolver muitos níveis de nodos de plano, mas a abordagem geral é a mesma: cada nodo computa e retorna sua próxima linha de saída cada vez que é chamado. Cada nodo é responsável pela aplicação da seleção e das expressões de projeção atribuídas ao mesmo pelo planejador.

O mecanismo do executor é utilizado para avaliar todos os quatro tipos de comando SQL básicos: *SELECT*, *INSERT*, *UPDATE* e *DELETE*. Para o *SELECT*, o código de nível mais alto do executor somente precisa enviar para o cliente cada linha retornada pelo plano de comando. Para o *INSERT*, cada linha retornada é inserida na tabela de destino especificada para o *INSERT* (Um comando *INSERT ... VALUES* cria uma árvore de plano trivial, consistindo de um único nodo *Result* que computa apenas uma linha de resultado, mas o comando *INSERT ... SELECT* pode demandar todo o poder do mecanismo do executor). Para o *UPDATE*, o planejador faz com que cada linha computada inclua os valores de todas as colunas atualizadas, mais o *TID* (ID da tupla, ou ID da linha) da linha de destino original; o nível superior do executor utiliza esta informação para criar uma nova linha atualizada, e marcar a linha antiga como excluída. Para o *DELETE*, a única coluna realmente retornada pelo plano é a *TID*, e o nível superior do executor simplesmente utiliza o *TID* para visitar cada linha de destino e marcá-la como excluída.



## Notas

1. **sintaxe** — do Lat. *syntaxe* < Gr. *śyntaxis*, arranjo, disposição — parte da estrutura gramatical de uma língua que contém as regras relativas à combinação das palavras em unidades maiores (como as orações), e as relações existentes entre as palavras dentro dessas unidades. PRIBERAM - Língua Portuguesa On-Line (<http://www.priberam.pt/dlpo/dlpo.aspx>). (N. do T.)
2. **léxico** — do Gr. *lěxicon*, relativo às palavras — dicionário de línguas clássicas antigas; dicionário abreviado; conjunto dos vocábulos de uma língua; dicionário dos vocábulos usados num domínio especializado (ciência, técnica). PRIBERAM - Língua Portuguesa On-Line (<http://www.priberam.pt/dlpo/dlpo.aspx>). (N. do T.)
3. **gramática** — do Lat. *grammatica* < *grammatike* — estudo ou tratado dos fatos da linguagem falada e escrita e das leis que a regulam; livro que contém as regras e os princípios que regem o funcionamento de uma língua; PRIBERAM - Língua Portuguesa On-Line (<http://www.priberam.pt/dlpo/dlpo.aspx>). (N. do T.)
4. **semântica** — do Gr. *semantiké*, da significação — estudo da linguagem humana do ponto de vista do significado das palavras e dos enunciados. PRIBERAM - Língua Portuguesa On-Line (<http://www.priberam.pt/dlpo/dlpo.aspx>). (N. do T.)

# Capítulo 41. Catálogos do sistema

Os catálogos do sistema são os locais onde os sistemas gerenciadores de banco de dados relacionais armazenam os metadados do esquema, tais como informações sobre tabelas e colunas, e informações de controle internas. Os catálogos do sistema do PostgreSQL são tabelas comuns. Estas tabelas podem ser removidas e recriadas, podem ser adicionadas colunas, podem ser inseridos e atualizados valores, e desta forma arruinar inteiramente o sistema. Normalmente, os catálogos do sistema não devem ser modificados manualmente, sempre existe um comando SQL para fazê-lo (Por exemplo, `CREATE DATABASE` insere uma linha no catálogo `pg_database` — e cria realmente o banco de dados no disco). Existem exceções para algumas operações obscuras muito particulares, como adicionar método de acesso de índice.<sup>1</sup>

## 41.1. Visão geral

A Tabela 41-1 lista os catálogos do sistema. A documentação mais detalhada sobre cada catálogo é mostrada a seguir.

Os catálogos do sistema são, em sua maioria, copiados do banco de dados modelo durante a criação do banco de dados e, portanto, estes catálogos são específicos do banco de dados. Alguns poucos catálogos são compartilhados fisicamente entre todos os bancos de dados do agrupamento; isto está indicado na descrição individual do catálogo.

**Tabela 41-1. Catálogos do sistema**

Nome do catálogo	Finalidade
<code>pg_aggregate</code>	funções de agregação
<code>pg_am</code>	métodos de acesso de índice
<code>pg_amop</code>	operadores de método de acesso
<code>pg_amproc</code>	procedimentos de suporte de método de acesso
<code>pg_attrdef</code>	valor padrão das colunas
<code>pg_attribute</code>	colunas de tabela (“atributos”)
<code>pg_cast</code>	casts (conversões de tipos de dado)
<code>pg_class</code>	tabelas, índices, seqüências, visões (“relações”)
<code>pg_constraint</code>	restrições de verificação, restrições de unicidade, restrições de chave primária, restrições de chave estrangeira
<code>pg_conversion</code>	informações sobre conversão de codificação
<code>pg_database</code>	bancos de dados que fazem parte deste agrupamento de bancos de dados
<code>pg_depend</code>	dependências entre objetos do banco de dados
<code>pg_description</code>	descrições ou comentários sobre os objetos do banco de dados
<code>pg_group</code>	grupos de usuários do banco de dados
<code>pg_index</code>	informações adicionais sobre índices
<code>pg_inherits</code>	hierarquia de herança de tabela
<code>pg_language</code>	linguagens para escrever funções
<code>pg_largeobject</code>	objetos grandes
<code>pg_listener</code>	suporte a notificação assíncrona
<code>pg_namespace</code>	esquemas

Nome do catálogo	Finalidade
pg_opclass	classes de operador de método de acesso de índice
pg_operator	operadores
pg_proc	funções e procedimentos
pg_rewrite	regras de reescrita dos comandos
pg_shadow	usuários do banco de dados
pg_statistic	estatísticas do planejador
pg_tablespace	espaços de tabelas do agrupamento de bancos de dados
pg_trigger	gatilhos
pg_type	tipos de dado

## 41.2. pg\_aggregate

O catálogo `pg_aggregate` armazena informações sobre funções de agregação. Uma função de agregação é uma função que opera sobre um conjunto de valores (tipicamente uma coluna de cada linha que corresponde à condição do comando), e retorna um único valor calculado a partir destes valores. As funções de agregação típicas são `sum`, `count` e `max`. Cada entrada em `pg_aggregate` é uma extensão de uma entrada em `pg_proc`. A entrada em `pg_proc` contém o nome da agregação, os tipos de dado de entrada e de saída, além de outras informações semelhantes às das funções comuns.

**Tabela 41-2. Colunas de `pg_aggregate`**

Nome	Tipo	Referencia	Descrição
aggfnoid	regproc	pg_proc.oid	OID da função de agregação em <code>pg_proc</code>
aggtransfn	regproc	pg_proc.oid	Função de transição da agregação
aggfinalfn	regproc	pg_proc.oid	Função final da agregação (zero se não houver nenhuma)
aggtranstype	oid	pg_type.oid	Tipo de dado da transição interna (estado) da função de agregação
agginitval	text		Valor inicial do estado da transição. É um campo texto, contendo o valor inicial na sua representação externa na forma de cadeia de caracteres. Se o valor for nulo, o valor do estado da transição começa por nulo.

As novas funções de agregação são registradas através do comando `CREATE AGGREGATE`. Para obter informações adicionais sobre como escrever funções de agregação, e o significado das funções de transição, etc., deve ser consultada a Seção 31.10

## 41.3. pg\_am

O catálogo `pg_am` armazena informações sobre os métodos de acesso de índice. Existe uma linha para cada método de acesso de índice suportado pelo sistema.

**Tabela 41-3. Colunas de `pg_am`**

Nome	Tipo	Referencia	Descrição
amname	name		Nome do método de acesso
amowner	int4	pg_shadow.usesysid	ID de usuário do dono (atualmente não utilizado)
amstrategies	int2		Número de estratégias do operador para este método de acesso

Nome	Tipo	Referencia	Descrição
amsupport	int2		Número de rotinas de suporte para este método de acesso
amorderstrategy	int2		Zero se o índice não oferece ordem de classificação, senão o número da estratégia do operador de estratégia que descreve a ordem de classificação
amcanunique	bool		O método de acesso suporta índices únicos?
amcanmulticol	bool		O método de acesso suporta índices de várias colunas?
amindexnulls	bool		O método de acesso suporta entradas de índice nulas?
amconcurrent	bool		O método de acesso suporta atualizações simultâneas?
amgettupple	regproc	pg_proc.oid	Função “próxima tupla válida”
aminsert	regproc	pg_proc.oid	Função “inserir esta tupla”
ambeginscan	regproc	pg_proc.oid	Função “começar nova varredura”
amrescan	regproc	pg_proc.oid	Função “recomeçar esta varredura”
amendscan	regproc	pg_proc.oid	Função “terminar esta varredura”
ammarkpos	regproc	pg_proc.oid	Função “marcar a posição corrente da varredura”
amrestrpos	regproc	pg_proc.oid	Função “restaurar a posição de varredura marcada”
ambuild	regproc	pg_proc.oid	Função “construir novo índice”
ambulkdelete	regproc	pg_proc.oid	Função “apagar em massa” (bulk-delete)
amvacuumcleanup	regproc	pg_proc.oid	Função de limpeza pós-VACUUM
amcostestimate	regproc	pg_proc.oid	Função para estimar o custo da varredura de índice

Um método de acesso de índice que suporta várias colunas (possui `amcanmulticol` verdade) *deve* suportar a indexação de valores nulos nas colunas após a primeira, porque o planejador assume que o índice pode ser utilizado para comandos acessando apenas a(s) primeira(s) coluna(s). Por exemplo, considere um índice em (a,b) e um comando com `WHERE a = 4`. O sistema assume que o índice pode ser utilizado para ser varrido à procura de linhas com `a = 4`, o que estaria errado se o índice omitisse as linhas onde `b` é nulo. Entretanto, não há problema se o índice omitir as linhas onde a primeira coluna indexada for nula (atualmente o GiST faz assim). `amindexnulls` deve ser definida como verdade somente se o método de acesso do índice indexar todas as linhas, incluindo combinações arbitrárias de valores nulos.

#### 41.4. pg\_amop

O catálogo `pg_amop` armazena informações sobre operadores associados a classes de operadores de método de acesso de índice. Existe uma linha para cada operador que é membro de uma classe de operadores.

**Tabela 41-4. Colunas de `pg_amop`**

Nome	Tipo	Referencia	Descrição
amopclaid	oid	pg_opclass.oid	Para que classe de operador de índice esta entrada se destina
amopsubtype	oid	pg_type.oid	Subtipo para distinguir várias entradas para a mesma estratégia; zero por padrão
amopstrategy	int2		Número da estratégia do operador
amopreqcheck	bool		Acerto no índice deve ser verificado novamente
amopopr	oid	pg_operator.oid	OID do operador

## 41.5. pg\_amproc

O catálogo `pg_amproc` armazena informações sobre os procedimentos de suporte associados às classes de operadores de método de acesso de índice. Existe uma linha para cada procedimento de suporte que pertence a uma classe de operadores.

**Tabela 41-5. Colunas de `pg_amproc`**

Nome	Tipo	Referencia	Descrição
<code>amopclaid</code>	<code>oid</code>	<code>pg_opclass.oid</code>	Para que classe de operador de índice esta entrada se destina
<code>amprocsubtype</code>	<code>oid</code>	<code>pg_type.oid</code>	Subtipo, se for uma rotina de tipo-cruzado, senão zero
<code>amprocnum</code>	<code>int2</code>		Número do procedimento de suporte
<code>amproc</code>	<code>regproc</code>	<code>pg_proc.oid</code>	OID do procedimento

## 41.6. pg\_attrdef

O catálogo `pg_attrdef` armazena o valor padrão das colunas. As principais informações sobre as colunas estão armazenadas em `pg_attribute` (veja abaixo). Somente as colunas que especificam explicitamente o valor padrão (quando a tabela é criada ou a coluna é adicionada) possuem uma entrada nesta tabela.

**Tabela 41-6. Colunas de `pg_attrdef`**

Nome	Tipo	Referencia	Descrição
<code>adrelid</code>	<code>oid</code>	<code>pg_class.oid</code>	Tabela que esta coluna pertence
<code>adnum</code>	<code>int2</code>	<code>pg_attribute.attnum</code>	Número da coluna
<code>adbin</code>	<code>text</code>		Representação interna do valor padrão da coluna
<code>adsrc</code>	<code>text</code>		Representação humanamente legível do valor padrão

O campo `adsrc` é histórico, sendo melhor não usá-lo, porque não acompanha mudanças externas que podem afetar a representação do valor padrão. A compilação reversa do campo `adbin` (com `pg_get_expr`, por exemplo) é uma forma melhor de mostrar o valor padrão.

## 41.7. pg\_attribute

O catálogo `pg_attribute` armazena informações sobre as colunas das tabelas. Existe exatamente uma linha em `pg_attribute` para cada coluna de cada tabela do banco de dados (Também existem entradas de atributo para os índices, e todos os objetos que possuem entrada em `pg_class`).

O termo atributo equivale a coluna, sendo usado por motivos históricos.

**Tabela 41-7. Colunas de `pg_attribute`**

Nome	Tipo	Referencia	Descrição
<code>attrelid</code>	<code>oid</code>	<code>pg_class.oid</code>	Tabela que esta coluna pertence
<code>attname</code>	<code>name</code>		Nome da coluna
<code>atttypid</code>	<code>oid</code>	<code>pg_type.oid</code>	Tipo de dado desta coluna
<code>attstattarget</code>	<code>int4</code>		<code>attstattarget</code> controla o nível de detalhe das estatísticas acumuladas para esta coluna pelo comando <i>ANALYZE</i> . O valor zero indica que não deve ser coletada nenhuma estatística. Um valor negativo diz para usar o padrão do sistema para estatísticas. O significado exato dos valores positivos é dependente do tipo de dado. Para os tipos de dado escalares, <code>attstattarget</code> é tanto o número de “valores mais comuns” a serem

Nome	Tipo	Referencia	Descrição
			coletados, quanto o número de barras do histograma a serem criadas
attlen	int2		Cópia de <code>pg_type.typelen</code> do tipo desta coluna
attnum	int2		Número da coluna. As colunas comuns são numeradas a partir de um. As colunas do sistema, tal como <code>oid</code> , possuem números negativos (arbitrários)
attndims	int4		Número de dimensões, se a coluna for de um tipo matriz; senão zero (Atualmente o número de dimensões de uma matriz não é imposto, portanto qualquer valor diferente de zero significa de fato “isto é uma matriz”)
attcacheoff	int4		Sempre -1 no armazenamento, mas quando carregado na memória em um descritor de linha pode ser atualizado para guardar o deslocamento do atributo na linha.
atttypmod	int4		<code>atttypmod</code> registra dados específicos do tipo, fornecidos na hora da criação da tabela (por exemplo, o comprimento máximo de uma coluna <code>varchar</code> ). É passado para as funções de entrada específicas dos tipos, e para as funções de imposição de comprimento. Geralmente o valor é -1 para os tipos que não necessitam de <code>atttypmod</code>
attbyval	bool		Cópia de <code>pg_type.typbyval</code> do tipo desta coluna
attstorage	char		Normalmente uma cópia de <code>pg_type.typstorage</code> do tipo desta coluna. Para os tipos de dado fatiáveis (TOAST), pode ser alterado após a criação da coluna para controlar a política de armazenamento
attalign	char		Cópia de <code>pg_type.typalign</code> do tipo desta coluna
attnotnull	bool		Representa uma restrição de não-nulo. É possível modificar esta coluna para habilitar ou desabilitar a restrição
atthasdef	bool		Indica se esta coluna possui um valor padrão e, neste caso, existe uma entrada correspondente no catálogo <code>pg_attrdef</code> que realmente define o valor
attisdropped	bool		Indica se esta coluna foi removida e, portanto, não é mais válida. Uma coluna removida continua presente na tabela, mas é ignorada pelo analisador e, portanto, não pode ser acessada via SQL
attislocal	bool		Indica se a coluna é definida localmente na relação. Deve ser observado que a coluna pode ser definida localmente e herdada simultaneamente.
attinhcount	int4		Número de ancestrais diretos que a coluna possui. Uma coluna com um número de ancestrais diferente de zero não pode ser removida nem renomeada.

Na entrada em `pg_attribute` da coluna removida, `atttypid` é redefinido como zero, mas `attlen` e os outros campos copiados de `pg_type` ainda são válidos. Esta arrumação é necessária para lidar com a situação onde o tipo de dado da coluna removida foi removido posteriormente, e portanto não existe mais a linha em `pg_type`. `attlen` e os outros campos podem ser utilizados para interpretar o conteúdo da linha da tabela.

## 41.8. pg\_cast

O catálogo `pg_cast` armazena caminhos de conversão de tipo de dados, tanto os caminhos nativos quanto os definidos através de `CREATE CAST`.

Tabela 41-8. Colunas de `pg_cast`

Nome	Tipo	Referencia	Descrição
<code>castsource</code>	<code>oid</code>	<code>pg_type.oid</code>	OID do tipo de dado de origem
<code>casttarget</code>	<code>oid</code>	<code>pg_type.oid</code>	OID do tipo de dado de destino
<code>castfunc</code>	<code>oid</code>	<code>pg_proc.oid</code>	OID da função a ser utilizada para realizar esta conversão. Zero se os tipos de dado forem binariamente compatíveis (ou seja, não é necessária nenhuma operação em tempo de execução para realizar a conversão).
<code>castcontext</code>	<code>char</code>		Indica em que contextos a conversão pode ser chamada. O valor <code>e</code> significa somente como uma conversão explícita (utilizando a sintaxe <code>CAST</code> ou <code>::</code> ). O valor <code>a</code> significa implicitamente na atribuição à coluna de destino, bem como explicitamente. O valor <code>i</code> significa implicitamente em expressões, bem como nos demais casos.

As funções de conversão listadas em `pg_cast` devem sempre receber o tipo de dado de origem como seu primeiro argumento, e retornar o tipo de destino da conversão como o tipo de seu resultado. Uma função de conversão pode ter até três argumentos. O segundo argumento, se estiver presente, deve ser do tipo `integer`; recebe o modificador de tipo associado ao tipo de destino, ou `-1` caso este não exista. O terceiro argumento, se estiver presente, deve ser do tipo `boolean`; recebe `true` se a conversão for uma conversão explícita, `false` caso contrário.

É legítimo criar entradas em `pg_cast` onde os tipos de origem e de destino sejam o mesmo, se estiver associada a uma função que recebe mais de um argumento. Estas entradas representam as “funções de imposição de comprimento”, que fazem os valores do tipo respeitar um determinado valor modificador de tipo. Entretanto, deve ser observado que no momento não há suporte para associar modificadores de tipo não-padrão a tipos de dado criados pelo usuário e, portanto, esta funcionalidade é para uso de apenas um pequeno número de tipos nativos que possuem uma sintaxe de modificador de tipo construída na gramática.

Quando a entrada em `pg_cast` possui tipos de dado de entrada e destino diferentes, e uma função que recebe mais de um argumento, esta entrada representa a conversão de um tipo para o outro e a aplicação da imposição do comprimento em um único passo. Quando não existe uma entrada deste tipo, a imposição para um tipo que utiliza um modificador de tipo envolve dois passos, um para converter entre os tipos de dados, e um segundo passo para aplicar o modificador.

## 41.9. `pg_class`

O catálogo `pg_class` cataloga as tabelas e tudo mais que possui colunas, ou é de alguma forma semelhante a uma tabela. Isto inclui os índices (mas consulte também `pg_index`), seqüências, visões, tipos compostos, e alguns tipos de relação especiais; consulte `relkind`. Abaixo, para nos referirmos a todos estes tipos de objetos falamos “relações”. Nem todas as colunas possuem significado em todos os tipos de relação.

Tabela 41-9. Colunas de `pg_class`

Nome	Tipo	Referencia	Descrição
<code>relname</code>	<code>name</code>		Nome da tabela, índice, visão, etc.
<code>relnamespace</code>	<code>oid</code>	<code>pg_namespace.oid</code>	OID do espaço de nomes que contém esta relação.
<code>reltype</code>	<code>oid</code>	<code>pg_type.oid</code>	OID do tipo de dado que corresponde ao tipo linha desta tabela, caso haja algum (zero para índices, que não possuem entrada em <code>pg_type</code> ).
<code>relowner</code>	<code>int4</code>	<code>pg_shadow.usesysid</code>	Dono da relação
<code>relam</code>	<code>oid</code>	<code>pg_am.oid</code>	Se for um índice, o método de acesso usado (B-tree, hash, etc.)
<code>relfilenode</code>	<code>oid</code>		Nome do arquivo em disco desta relação; 0 se não houver nenhum

Nome	Tipo	Referencia	Descrição
reltablespace	oid	pg_tablespace.oid	O espaço de tabelas onde a relação está armazenada. Se for zero, implica no espaço de tabelas padrão do banco de dados (Sem significado se a relação não possuir arquivo em disco).
relpages	int4		Tamanho da representação em disco desta tabela em páginas (com tamanho BLCKSZ). Esta é apenas uma estimativa utilizada pelo planejador. Atualizado pelos comandos VACUUM, ANALYZE e uns poucos comandos de DLL como o CREATE INDEX.
reltuples	float4		Número de linhas na tabela. Esta é apenas uma estimativa utilizada pelo planejador. Atualizado pelos comandos VACUUM, ANALYZE e uns poucos comandos de DLL como o CREATE INDEX.
reltoastrelid	oid	pg_class.oid	OID da tabela TOAST associada a esta tabela, 0 se não houver nenhuma. A tabela TOAST armazena atributos grandes “fora de linha”, em uma tabela secundária.
reltoastidxid	oid	pg_class.oid	Para a tabela TOAST, o OID de seu índice. 0 se não for uma tabela TOAST.
relhasindex	bool		Verdade se for uma tabela e possui (ou possuía recentemente) algum índice. É definido por CREATE INDEX, mas não é limpo imediatamente por DROP INDEX. O comando VACUUM limpa relhasindex quando descobre que a tabela não possui índices.
relisshared	bool		Verdade se esta tabela for compartilhada entre todos os bancos de dados do agrupamento. Somente alguns catálogos do sistema, como pg_database, são compartilhados
relkind	char		r = tabela comum, i = índice, s = seqüência, v = visão, c = tipo composto, s = especial, t = tabela TOAST
relnatts	int2		Número de colunas de usuário na relação (colunas do sistema não são contadas). Deve haver uma quantidade de entradas correspondente em pg_attribute. Consulte também pg_attribute.attnum.
relchecks	int2		Número de restrições de verificação na tabela; consulte o catálogo pg_constraint.
reltriggers	int2		Número de gatilhos na tabela; consulte o catálogo pg_trigger.
relukeys	int2		Não utilizado (Não é o número de chaves únicas)
relfkeys	int2		Não utilizado (Não é o número de chaves estrangeiras na tabela)
relrefs	int2		Não utilizado
relhasoids	bool		Verdade se for gerado um OID para cada linha da relação.
relhaspkey	bool		Verdade se a tabela possui (ou já possuiu) uma chave primária.
relhasrules	bool		Verdade se a tabela possui regras; consulte o catálogo



Nome	Tipo	Referencia	Descrição
			<code>pg_rewrite</code> .
<code>relhassubclass</code>	<code>bool</code>		Verdade se a tabela possui (ou possuiu) uma tabela que herda desta.
<code>relacl</code>	<code>aclitem[]</code>		Privilégios de acesso; consulte os comandos <i>GRANT</i> e <i>REVOKE</i> para obter detalhes.

## 41.10. `pg_constraint`

O catálogo `pg_constraint` armazena restrições de verificação, chave primária, unicidade e chave estrangeira em tabelas (As restrições de coluna não são tratadas de forma especial, porque toda restrição de coluna equivale a alguma restrição de tabela). As restrições de não-nulo são representadas no catálogo `pg_attribute`.

As restrições de verificação em domínios também são armazenadas neste catálogo.

**Tabela 41-10.** Colunas de `pg_constraint`

Nome	Tipo	Referencia	Descrição
<code>conname</code>	<code>name</code>		Nome da restrição (não necessariamente única!)
<code>connamespace</code>	<code>oid</code>	<code>pg_namespace.oid</code>	OID do espaço de nomes que contém a restrição.
<code>contype</code>	<code>char</code>		<code>c</code> = restrição de verificação; <code>f</code> = restrição de chave estrangeira; <code>p</code> = restrição de chave primária; <code>u</code> = restrição de unicidade.
<code>condeferrable</code>	<code>bool</code>		A restrição é postergável?
<code>condeferred</code>	<code>bool</code>		A restrição é postergada por padrão?
<code>conrelid</code>	<code>oid</code>	<code>pg_class.oid</code>	Tabela onde esta restrição se encontra; 0 zero se não for uma restrição de tabela
<code>contypid</code>	<code>oid</code>	<code>pg_type.oid</code>	Domínio onde esta restrição se encontra; 0 se não for uma restrição de domínio
<code>confrelid</code>	<code>oid</code>	<code>pg_class.oid</code>	Se for uma chave estrangeira, a tabela referenciada; senão 0
<code>confupdtype</code>	<code>char</code>		Código da ação de atualização da chave estrangeira
<code>confdeltype</code>	<code>char</code>		Código da ação de exclusão da chave estrangeira
<code>confmatchtype</code>	<code>char</code>		Tipo de correspondência da chave estrangeira
<code>conkey</code>	<code>int2[]</code>	<code>pg_attribute.attnum</code>	Se for uma restrição de tabela, a lista de colunas que a restrição abrange
<code>confkey</code>	<code>int2[]</code>	<code>pg_attribute.attnum</code>	Se for uma chave estrangeira, a lista de colunas referenciadas
<code>conbin</code>	<code>text</code>		Se for uma restrição de verificação, a representação interna da expressão
<code>consrc</code>	<code>text</code>		Se for uma restrição de verificação, a representação humanamente legível da expressão

**Nota:** `consrc` não é atualizada quando os objetos referenciados mudam; por exemplo, não acompanha a mudança dos nomes das colunas. Em vez de confiar neste campo, é melhor utilizar `pg_get_constraintdef()` para obter a definição da restrição de verificação.

**Nota:** `pg_class.relchecks` deve condizer com o número de entradas de restrição de verificação encontradas nesta tabela para uma determinada relação.

### 41.11. pg\_conversion

O catálogo `pg_conversion` descreve os procedimentos de conversão de codificação disponíveis. Para obter informações adicionais deve ser consultado o comando `CREATE CONVERSION`.

**Tabela 41-11. Colunas de `pg_conversion`**

Nome	Tipo	Referencia	Descrição
<code>conname</code>	<code>name</code>		Nome da conversão (único no espaço de nomes)
<code>connamespace</code>	<code>oid</code>	<code>pg_namespace.oid</code>	OID do espaço de nomes que contém esta conversão.
<code>conowner</code>	<code>int4</code>	<code>pg_shadow.usesysid</code>	Dono da conversão
<code>conforencoding</code>	<code>int4</code>		ID da codificação de origem
<code>contoencoding</code>	<code>int4</code>		ID da codificação de destino
<code>conproc</code>	<code>regproc</code>	<code>pg_proc.oid</code>	Procedimento de conversão
<code>condefault</code>	<code>bool</code>		Verdade se for a conversão padrão

### 41.12. pg\_database

O catálogo `pg_database` armazena informações sobre os bancos de dados disponíveis. Os bancos de dados são criados pelo comando `CREATE DATABASE`. Consulte o Capítulo 18 para obter detalhes sobre o significado de alguns parâmetros.

Diferentemente da maioria dos catálogos do sistema, `pg_database` é compartilhado por todos os bancos de dados do agrupamento: só existe uma instância de `pg_database` por agrupamento, e não uma por banco de dados.

**Tabela 41-12. Colunas de `pg_database`**

Nome	Tipo	Referencia	Descrição
<code>datname</code>	<code>name</code>		Nome do banco de dados
<code>datdba</code>	<code>int4</code>	<code>pg_shadow.usesysid</code>	Dono do banco de dados, geralmente o usuário que o criou
<code>encoding</code>	<code>int4</code>		Codificação dos caracteres deste banco de dados
<code>datistemplate</code>	<code>bool</code>		Se for verdade, então este banco de dados pode ser utilizado na cláusula <code>TEMPLATE</code> do comando <code>CREATE DATABASE</code> para criar um banco de dados novo que seja um clone deste.
<code>datallowconn</code>	<code>bool</code>		Se for falso, então ninguém pode se conectar a este banco de dados. É utilizado para proteger o banco de dados <code>template0</code> contra alterações.
<code>datlastsysoid</code>	<code>oid</code>		Último OID de sistema no banco de dados; particularmente útil para <code>pg_dump</code> .
<code>datvacuumxid</code>	<code>xid</code>		Todas as linhas inseridas ou excluídas por IDs de transação anteriores a este, foram marcadas como sabidamente efetivadas ou sabidamente interrompidas, neste banco de dados. É utilizado para determinar quando o espaço de <code>log</code> de efetivação pode ser reciclado.
<code>datfrozenxid</code>	<code>xid</code>		Todas as linhas inseridas por IDs de transação anteriores a este receberam um ID de transação permanente neste banco de dados (“foram congeladas”). Útil para verificar há necessidade de executar em breve o comando <code>VACUUM</code> no banco de dados para evitar o problema de reinício de ID de

Nome	Tipo	Referencia	Descrição
			transação.
dattablespace	oid	pg_tablespace.oid	O espaço de tabelas padrão para o banco de dados. Neste banco de dados, todas as tabelas para as quais <code>pg_class.reltablespace</code> for zero serão armazenadas neste espaço de tabelas; em particular, todos os catálogos do sistema não compartilhados se encontram neste espaço de tabelas.
datconfig	text[]		Padrões de sessão para as variáveis de configuração em tempo de execução
datacl	aclitem[]		Privilégios de acesso; para obter detalhes devem ser consultados os comandos <i>GRANT</i> e <i>REVOKE</i> .

### 41.13. pg\_depend

O catálogo `pg_depend` registra os relacionamentos de dependência entre os objetos do banco de dados. Esta informação permite ao comando `DROP` descobrir quais outros objetos devem ser removidos pelo `DROP CASCADE`, ou impedir a remoção no caso de `DROP RESTRICT`.

**Tabela 41-13. Colunas de `pg_depend`**

Nome	Tipo	Referencia	Descrição
classid	oid	pg_class.oid	OID do catálogo do sistema onde está o objeto dependente
objid	oid	qualquer coluna OID	OID do objeto dependente
objsubid	int4		Para uma coluna de tabela, o número da coluna (o <code>objid</code> e o <code>classid</code> se referem à própria tabela). Para todos os outros tipos de objeto esta coluna é zero.
refclassid	oid	pg_class.oid	OID do catálogo do sistema onde está o objeto referenciado
refobjid	oid	qualquer coluna OID	OID do objeto referenciado
refobjsubid	int4		Para uma coluna de tabela, o número da coluna (o <code>refobjid</code> e o <code>refclassid</code> se referem à própria tabela). Para todos os outros tipos de objeto esta coluna é zero.
deptype	char		Código definindo as semânticas específicas deste relacionamento de dependência; veja o texto.

Em todos os casos, um entrada em `pg_depend` indica que o objeto referenciado não pode ser removido sem que o objeto dependente também seja removido. Entretanto, existem diversas subcategorias identificadas por `deptype`:

#### DEPENDENCY\_NORMAL (n)

Relacionamento normal entre objetos criados separadamente. O objeto dependente pode ser removido sem afetar o objeto referenciado. O objeto referenciado só pode ser removido especificando `CASCADE` e, neste caso, o objeto dependente também é removido. Exemplo: uma coluna de tabela possui uma dependência normal de seu tipo de dado.

#### DEPENDENCY\_AUTO (a)

O objeto dependente pode ser removido separadamente do objeto referenciado, e deve ser removido automaticamente (a despeito do modo `RESTRICT` ou `CASCADE`) se o objeto referenciado for removido. Exemplo: uma restrição com nome em uma tabela é autodependente da tabela e, portanto, desaparece quando a tabela é removida.

**DEPENDENCY\_INTERNAL (i)**

O objeto dependente foi criado como parte da criação do objeto referenciado sendo, na verdade, apenas uma parte de sua implementação interna. O DROP do objeto dependente será desabilitado imediatamente (será informado ao usuário para executar o DROP do objeto referenciado, em vez deste objeto). O DROP do objeto referenciado é propagado para remover o objeto dependente, estando CASCADE especificado ou não. Exemplo: um gatilho criado para garantir a restrição de chave estrangeira é tornado internamente dependente da entrada em `pg_constraint` da restrição.

**DEPENDENCY\_PIN (p)**

Não existe nenhum objeto dependente; este tipo de entrada é um sinal que o próprio sistema depende do objeto referenciado e, portanto, o objeto não pode ser removido nunca. As entradas deste tipo são criadas apenas pelo `initdb`. As colunas do objeto dependente contêm zero.

Outras modalidades de dependência poderão ser necessárias no futuro.

## 41.14. pg\_description

O catálogo `pg_description` armazena descrições opcionais (comentários) para cada objeto do banco de dados. As descrições podem ser manipuladas pelo comando `COMMENT`, e vistas pelos comandos `\d` do `psql`. As descrições de vários objetos nativos do sistema estão presentes no conteúdo inicial de `pg_description`.

**Tabela 41-14. Colunas de `pg_description`**

Nome	Tipo	Referencia	Descrição
<code>objoid</code>	<code>oid</code>	qualquer coluna OID	OID do objeto que esta descrição pertence
<code>classoid</code>	<code>oid</code>	<code>pg_class.oid</code>	OID do catálogo do sistema onde este objeto se encontra
<code>objsubid</code>	<code>int4</code>		Para um comentário sobre uma coluna de tabela, é o número da coluna (o <code>objoid</code> e o <code>classoid</code> se referem à própria tabela). Para todos os outros tipos de objeto esta coluna é zero.
<code>description</code>	<code>text</code>		Texto arbitrário que serve como descrição do objeto

## 41.15. pg\_group

O catálogo `pg_group` define os grupos e armazena quais usuários pertencem a cada grupo. Os grupos são criados pelo comando `CREATE GROUP`. Consulte o Capítulo 17 para obter informações sobre o gerenciamento de privilégios.

Como as identidades dos usuários e dos grupos são para todo o agrupamento de bancos de dados, o catálogo `pg_group` é compartilhado por todos os bancos de dados do agrupamento: existe apenas uma instância de `pg_group` por agrupamento, e não uma por banco de dados.

**Tabela 41-15. Colunas de `pg_group`**

Nome	Tipo	Referencia	Descrição
<code>groname</code>	<code>name</code>		Nome do grupo
<code>grosysid</code>	<code>int4</code>		Número arbitrário para identificar o grupo
<code>grolist</code>	<code>int4[]</code>	<code>pg_shadow.usesysid</code>	Matriz contendo os IDs dos usuários neste grupo

## 41.16. pg\_index

O catálogo `pg_index` contém parte das informações sobre índices. O restante se encontra, em sua maioria, em `pg_class`.

Tabela 41-16. Colunas de `pg_index`

Nome	Tipo	Referencia	Descrição
<code>indexrelid</code>	<code>oid</code>	<code>pg_class.oid</code>	OID da entrada em <code>pg_class</code> para este índice
<code>indrelid</code>	<code>oid</code>	<code>pg_class.oid</code>	OID da entrada em <code>pg_class</code> da tabela a que este índice se destina
<code>indkey</code>	<code>int2vector</code>	<code>pg_attribute.attnum</code>	Matriz de valores <code>indnatts</code> (contendo até <code>INDEX_MAX_KEYS</code> ), indicando quais colunas da tabela este índice indexa. Por exemplo, o valor <code>1 3</code> significa que a primeira e a terceira coluna da tabela compõem a chave do índice. Um zero nesta matriz indica que o atributo do índice correspondente é uma expressão contendo colunas da tabela, em vez de uma simples referência à coluna.
<code>indclass</code>	<code>oidvector</code>	<code>pg_opclass.oid</code>	Contém o OID da classe de operadores a ser utilizada para cada coluna presente na chave do índice. Consulte <code>pg_opclass</code> para obter mais detalhes.
<code>indnatts</code>	<code>int2</code>		Número de colunas no índice (duplica <code>pg_class.relnatts</code> )
<code>indisunique</code>	<code>bool</code>		Se for verdade, então o índice é único
<code>indisprimary</code>	<code>bool</code>		Se for verdade, este índice representa a chave primária da tabela (A coluna <code>indisunique</code> deve ser sempre verdade quando esta coluna for verdade)
<code>indisclustered</code>	<code>bool</code>		Se for verdade, a tabela foi agrupada na última vez por este índice
<code>indexprs</code>	<code>text</code>		Árvores de expressão (na representação <code>nodeToString()</code> ) para os atributos do índice que não são simplesmente referências a colunas. É uma lista com um elemento para cada entrada igual a zero em <code>indkey</code> . Nulo se todos os atributos do índice são simplesmente referências a colunas.
<code>indpred</code>	<code>text</code>		Árvore de expressão (na representação <code>nodeToString()</code> ) para predicado de índice parcial. Nulo se não for um índice parcial

## 41.17. `pg_inherits`

O catálogo `pg_inherits` registra informações sobre hierarquias de herança de tabelas. Existe uma entrada para cada tabela diretamente descendente no banco de dados (As descendências indiretas podem ser determinadas seguindo a cadeia de entradas).

Tabela 41-17. Colunas de `pg_inherits`

Nome	Tipo	Referencia	Descrição
<code>inhrelid</code>	<code>oid</code>	<code>pg_class.oid</code>	OID da tabela descendente.
<code>inhparent</code>	<code>oid</code>	<code>pg_class.oid</code>	OID da tabela ancestral.
<code>inhseqno</code>	<code>int4</code>		Se houver mais de um ancestral direto para a tabela descendente (herança múltipla), este número informa a ordem pela qual as colunas herdadas devem ser dispostas. O contador começa por 1.

## 41.18. pg\_language

O catálogo `pg_language` registra as linguagens em que podem ser escritas as funções e procedimentos armazenados. Para obter informações adicionais sobre tratadores de linguagem deve ser consultado o comando `CREATE LANGUAGE` e o Capítulo 34.

**Tabela 41-18. Colunas de `pg_language`**

Nome	Tipo	Referencia	Descrição
<code>lanname</code>	<code>name</code>		Nome da linguagem
<code>lanispl</code>	<code>bool</code>		Falso para linguagens internas (tal como SQL), e verdade para as linguagens definidas pelo usuário. Atualmente o <code>pg_dump</code> ainda utiliza esta informação para determinar quais linguagens devem fazer parte da cópia de segurança, mas este mecanismo pode ser substituído por outro diferente alguma hora.
<code>lanpltrusted</code>	<code>bool</code>		Verdade se for uma linguagem confiável ( <code>trusted</code> ). Se for uma linguagem interna ( <code>lanispl</code> for falso), então esta coluna não tem sentido.
<code>lanplcallfoid</code>	<code>oid</code>	<code>pg_proc.oid</code>	Para as linguagens não internas é a referência ao tratador da linguagem, que é uma função especial responsável pela execução de todas as funções escritas nesta linguagem.
<code>lanvalidator</code>	<code>oid</code>	<code>pg_proc.oid</code>	Faz referência à função validadora da linguagem, responsável pela verificação da sintaxe e validação das novas funções quando estas são criadas. Zero se não for fornecida nenhuma função validadora.
<code>lanacl</code>	<code>aclitem[]</code>		Privilégios de acesso; para obter detalhes devem ser consultados os comandos <code>GRANT</code> e <code>REVOKE</code> .

## 41.19. pg\_largeobject

O catálogo `pg_largeobject` armazena os dados que constituem os “objetos grandes”. O objeto grande é identificado por um OID atribuído ao mesmo quando de sua criação. Cada objeto grande é fracionado em segmentos, ou “páginas”, pequenos o suficiente para serem convenientemente armazenados como linhas na catálogo `pg_largeobject`. A quantidade de dados por página é definida como sendo `LOBLKSIZE` (que atualmente é `BLCKSZ / 4` ou, tipicamente, 2 kB).

**Tabela 41-19. Colunas de `pg_largeobject`**

Nome	Tipo	Referencia	Descrição
<code>loid</code>	<code>oid</code>		Identificador do objeto grande que inclui esta página
<code>pageno</code>	<code>int4</code>		Número de página desta página dentro do objeto grande (contado a partir de zero)
<code>data</code>	<code>bytea</code>		Dados realmente armazenados no objeto grande. Nunca mais de <code>LOBLKSIZE</code> bytes, podendo ser menos.

Cada linha de `pg_largeobject` armazenas dados de uma página do objeto grande, começando pelo deslocamento de  $(pageno * LOBLKSIZE)$  bytes dentro do objeto. A implementação permite armazenamento esparsos: podem estar faltando páginas, e as páginas podem ter menos que `LOBLKSIZE` bytes, mesmo que não seja a última página do objeto. As regiões faltando dentro do objeto grande são lidas como zeros.

## 41.20. pg\_listener

O catálogo `pg_listener` dá suporte aos comandos `LISTEN` e `NOTIFY`. O ouvinte cria uma entrada em `pg_listener` para cada nome de notificação que está ouvindo. O notificador varre `pg_listener` e atualiza cada entrada correspondente

para mostrar que ocorreu uma notificação. O notificador também envia um sinal (utilizando o PID registrado na tabela) para acordar o ouvinte.

**Tabela 41-20. Colunas de `pg_listener`**

Nome	Tipo	Referencia	Descrição
<code>relname</code>	<code>name</code>		Nome da condição de notificação (O nome não precisa corresponder a nenhuma relação existente no banco de dados; o nome <code>relname</code> é histórico).
<code>listenerpid</code>	<code>int4</code>		PID do processo servidor que criou esta entrada
<code>notification</code>	<code>int4</code>		Zero se não houver nenhum evento pendente para este ouvinte. Havendo um evento pendente, o PID do processo servidor que enviou a notificação.

## 41.21. `pg_namespace`

O catálogo `pg_namespace` armazena espaços de nome. O espaço de nomes é a estrutura subjacente aos esquemas SQL: cada espaço de nomes pode ter uma coleção separada de relações, tipos, etc. sem conflito de nomes.

**Tabela 41-21. Colunas de `pg_namespace`**

Nome	Tipo	Referencia	Descrição
<code>nspname</code>	<code>name</code>		Nome do espaço de nomes
<code>nspowner</code>	<code>int4</code>	<code>pg_shadow.usesysid</code>	Dono do espaço de nomes
<code>nspacl</code>	<code>aclitem[]</code>		Privilégios de acesso; para obter detalhes devem ser consultados os comandos <i>GRANT</i> e <i>REVOKE</i> .

## 41.22. `pg_opclass`

O catálogo `pg_opclass` define as classes de operadores de método de acesso de índice. Cada classe de operadores define a semântica para as colunas do índice de um determinado tipo de dado e um determinado método de acesso de índice. Deve ser observado que podem existir várias classes de operadores para uma determinada combinação de tipo de dado/método de acesso e, assim, dando suporte a vários comportamentos.

As classes de operadores estão descritas por completo na Seção 31.14.

**Tabela 41-22. Colunas de `pg_opclass`**

Nome	Tipo	Referencia	Descrição
<code>opcamid</code>	<code>oid</code>	<code>pg_am.oid</code>	Classe de operadores de método de acesso de índice a que se destina
<code>opcname</code>	<code>name</code>		Nome desta classe de operadores
<code>opcnamespace</code>	<code>oid</code>	<code>pg_namespace.oid</code>	Espaço de nomes desta classe de operadores
<code>opcowner</code>	<code>int4</code>	<code>pg_shadow.usesysid</code>	Dono da classe de operadores
<code>opcintype</code>	<code>oid</code>	<code>pg_type.oid</code>	Tipo de dado que a classe de operadores indexa
<code>opcdefault</code>	<code>bool</code>		Verdade se esta classe de operadores for a classe padrão para <code>opcintype</code>
<code>opckeytype</code>	<code>oid</code>	<code>pg_type.oid</code>	Tipo de dado armazenado do índice, ou zero se for o mesmo que <code>opcintype</code>

A maior parte das informações que definem a classe de operadores na verdade não está na sua linha em `pg_opclass`, mas nas linhas associadas em `pg_amop` e `pg_amproc`. Estas linhas são consideradas como sendo parte da definição da classe de

operadores — não é diferente da maneira como uma relação é definida por uma única linha em `pg_class` mais as linhas associadas em `pg_attribute` e outras tabelas.

### 41.23. `pg_operator`

O catálogo `pg_operator` armazena informações sobre operadores. Para obter informações adicionais deve ser consultado o comando `CREATE OPERATOR` e a Seção 31.12.

**Tabela 41-23. Colunas de `pg_operator`**

Nome	Tipo	Referencia	Descrição
<code>oprname</code>	<code>name</code>		Nome do operador
<code>oprnamespace</code>	<code>oid</code>	<code>pg_namespace.oid</code>	OID do espaço de nomes que contém este operador.
<code>oprowner</code>	<code>int4</code>	<code>pg_shadow.usesysid</code>	Dono do operador
<code>oprkind</code>	<code>char</code>		<code>b</code> = infix (“ambos”), <code>l</code> = prefix (“esquerda”), <code>r</code> = postfix (“direita”)
<code>oprcanhash</code>	<code>bool</code>		Este operador suporta junções <code>hash</code>
<code>oprleft</code>	<code>oid</code>	<code>pg_type.oid</code>	Tipo do operando esquerdo
<code>oprright</code>	<code>oid</code>	<code>pg_type.oid</code>	Tipo do operando direito
<code>oprresult</code>	<code>oid</code>	<code>pg_type.oid</code>	Tipo do resultado
<code>oprcom</code>	<code>oid</code>	<code>pg_operator.oid</code>	Comutador deste operador, se houver algum
<code>oprnegate</code>	<code>oid</code>	<code>pg_operator.oid</code>	Negador deste operador, se houver algum
<code>oprlsortop</code>	<code>oid</code>	<code>pg_operator.oid</code>	Se este operador suportar junções por mesclagem ( <code>merge</code> ), o operador que classifica o tipo do operando à esquerda ( <code>L&lt;L</code> ).
<code>oprrsortop</code>	<code>oid</code>	<code>pg_operator.oid</code>	Se este operador suportar junções por mesclagem ( <code>merge</code> ), o operador que classifica o tipo do operando à direita ( <code>R&lt;R</code> ).
<code>oprltcmpop</code>	<code>oid</code>	<code>pg_operator.oid</code>	Se este operador suportar junções por mesclagem, o operador “menor-que” que compara os tipos dos operando à esquerda e à direita ( <code>L&lt;R</code> ).
<code>oprgtcmpop</code>	<code>oid</code>	<code>pg_operator.oid</code>	Se este operador suportar junções por mesclagem, o operador “maior-que” que compara os tipos dos operando à esquerda e à direita ( <code>L&gt;R</code> ).
<code>oprcode</code>	<code>regproc</code>	<code>pg_proc.oid</code>	Função que implementa este operador
<code>oprrest</code>	<code>regproc</code>	<code>pg_proc.oid</code>	Função estimadora de seletividade da restrição para este operador
<code>oprjoin</code>	<code>regproc</code>	<code>pg_proc.oid</code>	Função estimadora de seletividade da junção para este operador

As colunas não utilizadas contêm zero, por exemplo `oprleft` é zero para operadores de prefixo.

### 41.24. `pg_proc`

O catálogo `pg_proc` armazena informações sobre as funções (ou procedimentos). Para obter informações adicionais deve ser consultado o comando `CREATE FUNCTION` e a Seção 31.3.

A tabela contém dados para funções de agregação assim como para funções simples. Se `proisagg` for verdade, deve existir uma linha correspondente em `pg_aggregate`.



Tabela 41-24. Colunas de `pg_proc`

Nome	Tipo	Referencia	Descrição
<code>proname</code>	<code>name</code>		Nome da função
<code>pronamespace</code>	<code>oid</code>	<code>pg_namespace.oid</code>	OID do espaço de nomes que contém esta função.
<code>proowner</code>	<code>int4</code>	<code>pg_shadow.usesysid</code>	Dono da função
<code>prolang</code>	<code>oid</code>	<code>pg_language.oid</code>	Linguagem de implementação ou interface de chamada desta função
<code>proisagg</code>	<code>bool</code>		Verdade se a função for uma função de agregação
<code>prosecdef</code>	<code>bool</code>		Verdade se a função for uma função definidora de segurança (ou seja, uma função “setuid”)
<code>proisstrict</code>	<code>bool</code>		Verdade se a função retorna nulo quando algum argumento de chamada é nulo. Neste caso, na verdade, a função nem vai ser chamada. As funções que não são “estritas” devem estar preparadas para tratar entradas nulas.
<code>proretset</code>	<code>bool</code>		Verdade se a função retorna um conjunto (ou seja, vários valores do tipo de dado especificado)
<code>provolatile</code>	<code>char</code>		<code>provolatile</code> informa se o resultado da função depende apenas de seus argumentos de entrada, ou é afetada por fatores externos. Valor igual a <code>i</code> para as funções “imutáveis”, que sempre retornam o mesmo resultado para as mesmas entradas. Valor igual a <code>s</code> para as funções “estáveis”, cujos resultados (para entradas fixas) não mudam dentro de uma mesma varredura. Valor igual a <code>v</code> para as funções “voláteis”, cujos resultados podem mudar a qualquer instante (Também deve ser utilizado <code>v</code> para as funções com efeitos colaterais, para que as chamadas às mesmas não sejam otimizadas).
<code>pronargs</code>	<code>int2</code>		Número de argumentos
<code>prorettype</code>	<code>oid</code>	<code>pg_type.oid</code>	Tipo de dado do valor retornado
<code>proargtypes</code>	<code>oidvector</code>	<code>pg_type.oid</code>	Matriz contendo os tipos de dado dos argumentos da função
<code>proargnames</code>	<code>text[]</code>		Matriz contendo os nomes dos argumentos das funções. Os argumentos sem nome são definidos na matriz como cadeias de caracteres vazias. Se nenhum dos argumentos tiver nome, este campo pode ser nulo.
<code>prosrc</code>	<code>text</code>		Informa ao tratador da função como chamar a função. Podendo ser o código fonte da função para as linguagens interpretadas, o símbolo de ligação, o nome do arquivo, ou qualquer outra coisa, dependendo da linguagem de implementação/convenção de chamada.
<code>probin</code>	<code>bytea</code>		Informações adicionais sobre como chamar a função. Novamente, a interpretação é específica da linguagem.
<code>proacl</code>	<code>aclitem[]</code>		Privilégios de acesso; para obter detalhes devem ser consultados os comandos <i>GRANT</i> e <i>REVOKE</i> .

Para as funções compiladas, tanto nativas quanto carregadas dinamicamente, `prosrc` contém o nome da função na linguagem C (símbolo de ligação). Para todos os outros tipos de linguagem conhecidos no momento, `prosrc` contém o

texto do código fonte da função. `probin` não é utilizado, exceto para as funções C carregadas dinamicamente, para as quais fornece o nome do arquivo de biblioteca compartilhada contendo a função.

## 41.25. `pg_rewrite`

O catálogo `pg_rewrite` armazena regras de reescrita para tabelas e visões.

**Tabela 41-25. Colunas de `pg_rewrite`**

Nome	Tipo	Referencia	Descrição
<code>rulename</code>	<code>name</code>		Nome da regra
<code>ev_class</code>	<code>oid</code>	<code>pg_class.oid</code>	Tabela para a qual esta regra se destina
<code>ev_attr</code>	<code>int2</code>		Coluna para a qual esta regra se destina (atualmente sempre zero para indicar toda a tabela)
<code>ev_type</code>	<code>char</code>		Tipo de evento para o qual esta regra se destina: 1 = SELECT, 2 = UPDATE, 3 = INSERT, 4 = DELETE.
<code>is_instead</code>	<code>bool</code>		Verdade se a regra for uma regra <code>INSTEAD</code>
<code>ev_qual</code>	<code>text</code>		Árvore de expressão (na forma de uma representação <code>nodeToString()</code> ) para a condição de qualificação da regra.
<code>ev_action</code>	<code>text</code>		Árvore de comando (na forma de uma representação <code>nodeToString()</code> ) para a ação da regra.

**Nota:** `pg_class.relhasrules` deve ser verdade quando a tabela possui alguma regra neste catálogo.

## 41.26. `pg_shadow`

O catálogo `pg_shadow` contém informações sobre os usuários do banco de dados. Este nome vem do fato desta tabela não poder ser lida por todos, uma vez que contém senhas. `pg_user` é uma visão de `pg_shadow` que pode ser lida por todos, uma vez que esconde o campo senha.

O Capítulo 17 contém informações detalhadas sobre o gerenciamento de usuários e de privilégios.

Uma vez que as identidades dos usuários valem para todo o agrupamento, o catálogo `pg_shadow` é compartilhado por todos os bancos de dados do agrupamento: existe apenas uma instância de `pg_shadow` por agrupamento, e não uma por banco de dados.

**Tabela 41-26. Colunas de `pg_shadow`**

Nome	Tipo	Referencia	Descrição
<code>username</code>	<code>name</code>		Nome do usuário
<code>usesysid</code>	<code>int4</code>		Id do usuário (número arbitrário utilizado para referenciar este usuário)
<code>usecreatedb</code>	<code>bool</code>		Se for verdade o usuário pode criar bancos de dados
<code>usesuper</code>	<code>bool</code>		Se for verdade o usuário é um superusuário
<code>usecatupd</code>	<code>bool</code>		Se for verdade o usuário pode atualizar os catálogos do sistema (Mesmo os superusuários não podem atualizar os catálogos do sistema a não ser que esta coluna seja verdade).
<code>passwd</code>	<code>text</code>		Senha (possivelmente criptografada)
<code>valuntil</code>	<code>abstime</code>		Momento de expiração da conta (usado apenas para autenticação por senha)
<code>useconfig</code>	<code>text[]</code>		Padrões da sessão para as variáveis de configuração em tempo de execução

## 41.27. pg\_statistic

O catálogo `pg_statistic` armazena dados estatísticos sobre o conteúdo do banco de dados. As entradas são criadas pelo comando `ANALYZE` e depois utilizadas pelo planejador de comandos. Existe uma entrada para cada coluna de tabela que foi analisada. Deve ser observado que todos os dados estatísticos são inerentemente aproximados, mesmo assumindo que estejam atualizados.

O catálogo `pg_statistic` também armazena dados estatísticos sobre os valores das expressões de índice, que são descritos como se fossem colunas de dados reais; particularmente, `starelid` referencia o índice. Entretanto, não é criada nenhuma entrada para uma coluna de índice comum que não é uma expressão, uma vez que seria redundante com a coluna da tabela subjacente.

Uma vez que podem ser apropriados tipos de estatística diferentes para tipos de dado diferentes, o catálogo `pg_statistic` foi projetado para não assumir muita coisa sobre que tipo de estatística será armazenado. Somente estatísticas extremamente gerais (como nulidade) recebem colunas dedicadas em `pg_statistic`. Tudo mais é armazenado em “encaixes” (`slots`), que são grupos de colunas associadas cujo conteúdo é identificado por um código numérico em uma das colunas do encaixe. Para obter informações adicionais deve ser consultado o arquivo `src/include/catalog/pg_statistic.h`.

O catálogo `pg_statistic` não deve poder ser visto por todos, uma vez que mesmo informações estatísticas sobre o conteúdo da tabela podem ser consideradas confidenciais (Exemplo: os valores mínimo e máximo da coluna salário podem ser bastante interessantes). `pg_stats` é uma visão de `pg_statistic` que pode ser lida por todos, e mostra apenas informações sobre as tabelas que podem ser lidas pelo usuário corrente.

**Tabela 41-27. Colunas de `pg_statistic`**

Nome	Tipo	Referencia	Descrição
<code>starelid</code>	<code>oid</code>	<code>pg_class.oid</code>	Tabela ou índice que a coluna descrita pertence
<code>staattnum</code>	<code>int2</code>	<code>pg_attribute.attnum</code>	Número da coluna descrita
<code>stanullfrac</code>	<code>float4</code>		Fração das entradas nulas na coluna
<code>stawidth</code>	<code>int4</code>		Largura armazenada média, em bytes, das entradas não-nulas
<code>stadistinct</code>	<code>float4</code>		Número de valores de dado não-nulos distintos na coluna. Um valor maior do que zero é o número real de valores distintos. Um valor menor do que zero é o negativo da fração do número de linhas da tabela (por exemplo, uma coluna em que cada valor aparece duas vezes, em média, pode ser representada por <code>stadistinct = -0.5</code> ). O valor zero significa que o número de valores distintos é desconhecido.
<code>stakindN</code>	<code>int2</code>		Código numérico indicando o tipo de estatística armazenada no <i>N</i> -ésimo “encaixe” da linha de <code>pg_statistic</code> .
<code>staopN</code>	<code>oid</code>	<code>pg_operator.oid</code>	Operador utilizado para derivar as estatísticas armazenadas no <i>N</i> -ésimo “encaixe”. Por exemplo, em um encaixe tipo histograma é mostrado o operador <code>&lt;</code> que define a ordem de classificação dos dados.
<code>stanumbersN</code>	<code>float4[]</code>		Estatísticas numéricas do tipo apropriado para o <i>N</i> -ésimo “encaixe”, ou nulo se o tipo do encaixe não envolve valores numéricos.
<code>stavaluesN</code>	<code>anyarray</code>		Valores dos dados da coluna, do tipo apropriado, para o <i>N</i> -ésimo “encaixe”, ou nulo se o tipo do encaixe não armazena nenhum valor de dado. Cada valor do elemento da matriz é do tipo de dado específico da coluna, portanto não há como definir o tipo de dado desta coluna de forma mais específica que <code>anyarray</code> .

## 41.28. pg\_tablespace

O catálogo `pg_tablespace` armazena informações sobre os espaços de tabelas disponíveis. As tabelas podem ser colocadas em um determinado espaço de tabelas para ajudar administrar a organização dos discos.

Diferentemente da maioria dos catálogos do sistema, o catálogo `pg_tablespace` é compartilhado por todos os bancos de dados do agrupamento: existe apenas uma instância do catálogo `pg_tablespace` por agrupamento, e não uma por banco de dados.

**Tabela 41-28. Colunas de `pg_tablespace`**

Nome	Tipo	Referencia	Descrição
<code>spcname</code>	<code>name</code>		Tablespace name
<code>spcowner</code>	<code>int4</code>	<code>pg_shadow.usesysid</code>	Dono do espaço de tabelas, geralmente o usuário que o criou
<code>spclocation</code>	<code>text</code>		Local (caminho do diretório) do espaço de tabelas
<code>spcacl</code>	<code>aclitem[]</code>		Privilégios de acesso; para obter detalhes devem ser consultados os comandos <i>GRANT</i> e <i>REVOKE</i> .

## 41.29. pg\_trigger

O catálogo `pg_trigger` armazena os gatilhos das tabelas. Para obter informações adicionais deve ser consultado o comando *CREATE TRIGGER*.

**Tabela 41-29. Colunas de `pg_trigger`**

Nome	Tipo	Referencia	Descrição
<code>tgrelid</code>	<code>oid</code>	<code>pg_class.oid</code>	Tabela onde o gatilho se encontra
<code>tgname</code>	<code>name</code>		Nome do gatilho (deve ser único entre os gatilhos da mesma tabela)
<code>tgfoid</code>	<code>oid</code>	<code>pg_proc.oid</code>	Função a ser chamada
<code>tgtype</code>	<code>int2</code>		Máscara de bits identificando as condições do gatilho
<code>tgenabled</code>	<code>bool</code>		Verdade se o gatilho estiver habilitado (atualmente não é verificado em todos os lugares onde deveria ser, portanto desabilitar o gatilho definindo esta coluna como falso não funciona de forma confiável)
<code>tgisconstraint</code>	<code>bool</code>		Verdade se o gatilho implementa uma restrição de integridade referencial
<code>tgconstrname</code>	<code>name</code>		Nome da restrição de integridade referencial
<code>tgconstrrelid</code>	<code>oid</code>	<code>pg_class.oid</code>	Tabela referenciada pela restrição de integridade referencial
<code>tgdeferrable</code>	<code>bool</code>		Verdade se for postergável
<code>tginitdeferred</code>	<code>bool</code>		Verdade se for inicialmente postergado
<code>tgargs</code>	<code>int2</code>		Número de cadeias de caracteres argumentos passadas para a função de gatilho
<code>tgattr</code>	<code>int2vector</code>		Atualmente não é utilizado
<code>tgargs</code>	<code>bytea</code>		Cadeias de caracteres argumentos a serem passadas para o gatilho, todas terminadas por nulo

**Nota:** `pg_class.reltriggers` deve corresponder ao número de gatilhos encontrados nesta tabela para uma determinada relação.

### 41.30. `pg_type`

O catálogo `pg_type` armazena informações sobre tipos de dado. Os tipos base (tipos escalares) são criados pelo comando `CREATE TYPE`, e os domínios pelo comando `CREATE DOMAIN`. Para cada tabela do banco de dados é criado, automaticamente, um tipo composto para representar a estrutura da linha da tabela. Também é possível criar tipos compostos utilizando o comando `CREATE TYPE AS`.

**Tabela 41-30. Colunas de `pg_type`**

Nome	Tipo	Referencia	Descrição
<code>typname</code>	<code>name</code>		Nome do tipo de dado
<code>typnamespace</code>	<code>oid</code>	<code>pg_namespace.oid</code>	OID do espaço de nomes que contém este tipo.
<code>typowner</code>	<code>int4</code>	<code>pg_shadow.usesysid</code>	Dono do tipo
<code>typplen</code>	<code>int2</code>		Para um tipo de tamanho fixo, <code>typplen</code> é o número de bytes na representação interna do tipo. Para um tipo de tamanho variável, <code>typplen</code> é negativo. -1 indica um tipo “varlena” (aquele que tem a palavra comprimento), -2 indica uma cadeia de caracteres C terminada por nulo.
<code>typbyval</code>	<code>bool</code>		<code>typbyval</code> determina se as rotinas internas passam o valor deste tipo por valor ou por referência. É melhor <code>typbyval</code> ser falso se <code>typplen</code> não for igual a 1, 2 ou 4 (ou 8 nas máquinas com Datum igual a 8 bytes). Os tipos de comprimento variável são sempre passados por referência. Deve ser observado que <code>typbyval</code> pode ser falso mesmo quando o comprimento permite passar por valor; atualmente isto é verdade para o tipo <code>float4</code> , por exemplo.
<code>typtype</code>	<code>char</code>		<code>typtype</code> é igual a <code>b</code> para um tipo base, igual a <code>c</code> para um tipo composto (ou seja, o tipo de uma linha de tabela), igual a <code>d</code> para um domínio, ou igual a <code>p</code> para um pseudotipo. Consulte também <code>typrelid</code> e <code>typbasetype</code> .
<code>typisdefined</code>	<code>bool</code>		Verdade se o tipo está definido, falso se for apenas um espaço reservado para um tipo que ainda não está definido. Quando <code>typisdefined</code> é falso, não se pode confiar em nada além do nome do tipo, espaço de nomes e OID.
<code>typdelim</code>	<code>char</code>		Caractere que separa dois valores deste tipo ao analisar entrada na forma de matriz. Deve ser observado que o delimitador está associado ao tipo de dado do elemento, e não ao tipo de dado da matriz
<code>typrelid</code>	<code>oid</code>	<code>pg_class.oid</code>	Se for um tipo composto (consulte <code>typtype</code> ), então esta coluna aponta para a entrada em <code>pg_class</code> que define a tabela correspondente (Para um tipo composto independente, a entrada em <code>pg_class</code> não representa realmente uma tabela, mas de todo jeito é necessária para as entradas em <code>pg_attribute</code> do tipo fazerem ligação). Zero para os tipos não-compostos.
<code>typelem</code>	<code>oid</code>	<code>pg_type.oid</code>	Se <code>typelem</code> não for 0, então identifica outra linha em <code>pg_type</code> . O tipo corrente pode ser acessado como qualquer matriz produzindo valores do tipo <code>typelem</code> . Um tipo matriz

Nome	Tipo	Referencia	Descrição
			“de verdade” é de comprimento variável ( <code>typelen = -1</code> ), mas alguns tipos de comprimento fixo ( <code>typelen &gt; 0</code> ) também possuem <code>typelem</code> diferente de zero como, por exemplo, <code>name</code> e <code>oidvector</code> . Se um tipo de comprimento fixo possui <code>typelem</code> , então sua representação interna deve ser alguma quantidade de valores do tipo de dado <code>typelem</code> com nenhum outro dado. Os tipos matriz de comprimento variável possuem um cabeçalho definido pelas subrotinas da matriz.
<code>typinput</code>	<code>regproc</code>	<code>pg_proc.oid</code>	Função de conversão da entrada (formato texto)
<code>typoutput</code>	<code>regproc</code>	<code>pg_proc.oid</code>	Função de conversão da saída (formato texto)
<code>typreceive</code>	<code>regproc</code>	<code>pg_proc.oid</code>	Função de conversão da entrada (formato binário), ou 0 se nenhuma
<code>typsend</code>	<code>regproc</code>	<code>pg_proc.oid</code>	Função de conversão da saída (formato binário), ou 0 se nenhuma
<code>typanalyze</code>	<code>regproc</code>	<code>pg_proc.oid</code>	Função ANALYZE personalizada, ou 0 para utilizar a função padrão
<code>typalign</code>	<code>char</code>		<p><code>typalign</code> é o alinhamento requerido para armazenar um valor deste tipo. Se aplica ao armazenamento em disco, assim como a maioria das representações do valor dentro do PostgreSQL. Quando vários valores são armazenados consecutivamente, tal como na representação de uma linha completa no disco, é inserido um preenchimento antes do dado deste tipo para que comece na fronteira especificada. A referência de alinhamento é o início do primeiro dado na sequência.</p> <p>Os valores possíveis são:</p> <ul style="list-style-type: none"> <li>• <code>c</code> = alinhamento <code>char</code>, ou seja, nenhum alinhamento é necessário.</li> <li>• <code>s</code> = alinhamento <code>short</code> (2 bytes na maioria das máquinas).</li> <li>• <code>i</code> = alinhamento <code>int</code> (4 bytes na maioria das máquinas).</li> <li>• <code>d</code> = alinhamento <code>double</code> (8 bytes na maioria das máquinas, mas de forma alguma em todas).</li> </ul> <p><b>Nota:</b> Para os tipos utilizados nas tabelas do sistema, é crítico que o tamanho e o alinhamento definidos em <code>pg_type</code> estejam de acordo com a forma como o compilador organiza a coluna na estrutura que representa a linha da tabela.</p>
<code>typstorage</code>	<code>char</code>		<p><code>typstorage</code> informa para os tipos varlena (aqueles com <code>typelen = -1</code>) se o tipo está preparado para o fatiamento (<code>toasting</code>), e qual deve ser a estratégia padrão para os atributos deste tipo. Os valores possíveis são:</p> <ul style="list-style-type: none"> <li>• <code>p</code>: O valor deve ser sempre armazenado diretamente.</li> <li>• <code>e</code>: O valor pode ser armazenado em uma relação “secundária” (se a relação possuir uma, consulte <code>pg_class.reltoastrelid</code>).</li> <li>• <code>m</code>: O valor pode ser armazenado em-linha comprimido.</li> </ul>

Nome	Tipo	Referencia	Descrição
			<ul style="list-style-type: none"> <li>• <code>x</code>: O valor pode ser armazenado em-linha comprimido, ou posto no armazenamento “secundário”.</li> </ul> <p>Deve ser observado que as colunas do tipo <code>m</code> também podem ser movidas para o armazenamento secundário, mas somente como último recurso (Primeiro são movidas as colunas do tipo <code>e</code> e do tipo <code>x</code>).</p>
<code>typnotnull</code>	<code>bool</code>		<code>typnotnull</code> representa uma restrição de não-nulo no tipo. Utilizado apenas em domínios.
<code>typbasetype</code>	<code>oid</code>	<code>pg_type.oid</code>	Se for um domínio (consulte <code>typtype</code> ), então <code>typbasetype</code> identifica o tipo em que é baseado. Zero se não for um domínio.
<code>typtypmod</code>	<code>int4</code>		Os domínios utilizam <code>typtypmod</code> para registrar o <code>typmod</code> a ser aplicado ao seu tipo base (-1 se o tipo base não utilizar um <code>typmod</code> ). -1 se este tipo não for um domínio.
<code>typndims</code>	<code>int4</code>		<code>typndims</code> é o número de dimensões da matriz para o domínio que é uma matriz (ou seja, <code>typbasetype</code> é um tipo matriz; o <code>typelem</code> do domínio corresponde ao tipo base de <code>typelem</code> ). Zero para os tipos que não forem domínios matriz.
<code>typdefaultbin</code>	<code>text</code>		Se <code>typdefaultbin</code> não for nulo, é a representação <code>nodeToString()</code> da expressão padrão para o tipo. Utilizado apenas para domínios.
<code>typdefault</code>	<code>text</code>		<code>typdefault</code> é nulo se o tipo não tiver valor padrão associado. Se <code>typdefaultbin</code> não for nulo, <code>typdefault</code> deve conter uma versão humanamente legível da expressão padrão representada por <code>typdefaultbin</code> . Se <code>typdefaultbin</code> for nulo e <code>typdefault</code> não for, então <code>typdefault</code> é a representação externa do valor padrão do tipo, o qual pode ser introduzido no conversor de entrada do tipo para produzir uma constante.

### 41.31. Visões do sistema

Além dos catálogos do sistema, o PostgreSQL disponibiliza várias visões nativas. Algumas visões do sistema fornecem um acesso conveniente a algumas consultas aos catálogos do sistema utilizadas com frequência. Outras visões também fornecem acesso ao estado interno do servidor.

O esquema de informações (Capítulo 30) provê um conjunto alternativo de visões, que se sobrepõem às funcionalidades das visões do sistema. Uma vez que o esquema de informações faz parte do padrão SQL, enquanto as visões aqui descritas são específicas do PostgreSQL, geralmente é melhor utilizar o esquema de informações quando este fornece todas as informações necessárias.

A Tabela 41-31 lista as visões do sistema descritas neste capítulo. A documentação mais detalhadas de cada visão vem a seguir. Existem algumas visões adicionais que fornecem acesso aos resultados do coletor de estatísticas; estão descritas na Tabela 23-1.

Exceto quando indicado, todas as visões descritas aqui são somente para leitura.

**Tabela 41-31. Visões do sistema**

Nome da visão	Finalidade
pg_indexes	índices
pg_locks	bloqueios mantidos no momento
pg_rules	regras
pg_settings	configurações dos parâmetros
pg_stats	estatísticas do planejador
pg_tables	tabelas
pg_user	usuários do banco de dados
pg_views	visões

## 41.32. pg\_indexes

A visão `pg_indexes` fornece acesso a informações úteis sobre cada índice do banco de dados.

**Tabela 41-32. Colunas de `pg_indexes`**

Nome	Tipo	Referencia	Descrição
schemaname	name	<code>pg_namespace.nspname</code>	Nome do esquema que contém a tabela e o índice
tablename	name	<code>pg_class.relname</code>	Nome da tabela para a qual o índice se destina
indexname	name	<code>pg_class.relname</code>	Nome do índice
tablespace	name	<code>pg_tablespace.spcname</code>	Nome do espaço de tabelas contendo o índice (NULL se for o padrão para o banco de dados).
indexdef	text		Definição do índice (o comando de criação reconstruído)

## 41.33. pg\_locks

A visão `pg_locks` fornece acesso a informações sobre bloqueios dentro do servidor de banco de dados mantidos por transações em aberto. Veja no Capítulo 12 mais explicações sobre bloqueio.

`pg_locks` contém uma linha por objeto bloqueável ativo, modo de bloqueio requisitado e transação relevante. Assim, o mesmo objeto bloqueável pode aparecer várias vezes, se várias transações estiverem mantendo ou aguardando por bloqueios no mesmo. Entretanto, um objeto que atualmente não possui nenhum bloqueio atuando sobre o mesmo não aparece. Um objeto bloqueável é uma relação (por exemplo, uma tabela), ou um ID de transação.

Deve ser observado que esta visão inclui apenas bloqueios no nível de tabela, e não no nível de linha. Se a transação estiver aguardando por um bloqueio no nível de linha, é mostrada na visão como aguardando pelo ID da transação que está atualmente mantendo o bloqueio na linha.

**Tabela 41-33. Colunas de `pg_locks`**

Nome	Tipo	Referencia	Descrição
relation	oid	<code>pg_class.oid</code>	OID da relação bloqueada, ou NULL se o objeto bloqueável for um ID de transação.
database	oid	<code>pg_database.oid</code>	OID do banco de dados em que a relação bloqueada se encontra, ou zero se a relação bloqueada for uma tabela compartilhada globalmente, ou NULL se o objeto bloqueável for um ID de transação.



Nome	Tipo	Referencia	Descrição
transaction	xid		ID da transação, ou NULL se o objeto bloqueável for uma relação.
pid	integer		ID de processo do processo servidor mantendo ou aguardando pelo bloqueio
mode	text		Nome do modo de bloqueio mantido ou desejado por este processo (consulte a Seção 12.3.1)
granted	boolean		Verdade se o bloqueio está sendo mantido, falso se o bloqueio está sendo aguardado

`granted` é verdade em uma linha representando um bloqueio mantido pela sessão indicada. Falso indica que esta sessão está atualmente aguardando para obter o bloqueio, o que implica que alguma outra sessão está mantendo um modo de bloqueio conflitante no mesmo objeto bloqueável. A sessão aguardando dorme até que o bloqueio seja liberado( ou até que uma situação de impasse seja detectada). Uma única sessão pode estar aguardando para obter no máximo um único bloqueio por vez.

Cada transação mantém um bloqueio exclusivo em seu ID de transação por toda a sua duração. Se uma transação descobrir que é necessário aguardar especificamente por outra transação, ela fará isto tentando obter um bloqueio compartilhado no ID da outra transação. Somente será bem-sucedido quando a outra transação terminar e liberar seus bloqueios.

Quando a visão `pg_locks` é acessada, as estruturas de dado internas do gerenciador de bloqueios são momentaneamente bloqueadas, e é feita uma cópia para ser mostrada pela visão. Isto garante que a visão produz um conjunto de resultados consistente, enquanto que não bloqueia as operações normais do gerenciador de bloqueios por mais tempo que o necessário. Apesar disso, pode haver algum impacto no desempenho do banco de dados se esta visão for consultada frequentemente.

`pg_locks` provê uma visão global de todos os bloqueios no agrupamento de bancos de dados, e não apenas aqueles relevantes para o banco de dados corrente. Embora possa ser feita uma junção de sua coluna `relation` com `pg_class.oid` para identificar as relações bloqueadas, isto só funciona de forma correta para as relações no banco de dados corrente (aqueles em que a coluna `database` é o OID do banco de dados corrente ou zero).

Se o coletor de estatísticas estiver habilitado, pode ser feita a junção da coluna `pid` com a coluna `procpid` da visão `pg_stat_activity` para obter mais informações sobre a sessão mantendo ou aguardando para obter o bloqueio.

## 41.34. pg\_rules

A visão `pg_rules` fornece acesso a informações úteis sobre as regras de reescrita de comandos.

**Tabela 41-34. Colunas de `pg_rules`**

Nome	Tipo	Referencia	Descrição
schemaname	name	<code>pg_namespace.nspname</code>	Nome do esquema que contém a tabela
tablename	name	<code>pg_class.relname</code>	Nome da tabela para a qual esta regra se destina
rulename	name	<code>pg_rewrite.rulename</code>	Nome da regra
definition	text		Definição da regra (o comando de criação reconstruído)

A visão `pg_rules` exclui as regras ON SELECT da visão; estas podem ser vistas em `pg_views`.

## 41.35. pg\_settings

A visão `pg_settings` fornece acesso a parâmetros em tempo de execução do servidor. Essencialmente é uma interface alternativa aos comandos `SHOW` e `SET`. Também fornece acesso a alguns fatos sobre cada parâmetro não disponíveis diretamente no `SHOW`, como os valores mínimo e máximo.

Tabela 41-35. Colunas de `pg_settings`

Nome	Tipo	Referencia	Descrição
<code>name</code>	<code>text</code>		Nome do parâmetro de configuração em tempo de execução
<code>setting</code>	<code>text</code>		Valor corrente do parâmetro
<code>category</code>	<code>text</code>		Grupo lógico do parâmetro.
<code>short_desc</code>	<code>text</code>		Breve descrição do parâmetro.
<code>extra_desc</code>	<code>text</code>		Informação adicional, mais detalhada, sobre o parâmetro.
<code>context</code>	<code>text</code>		Contexto requerido para definir o valor do parâmetro.
<code>vartype</code>	<code>text</code>		Tipo do parâmetro ( <code>bool</code> , <code>integer</code> , <code>real</code> ou <code>string</code> ).
<code>source</code>	<code>text</code>		Origem do valor corrente do parâmetro
<code>min_val</code>	<code>text</code>		Valor mínimo permitido para o parâmetro (NULL para valores não numéricos)
<code>max_val</code>	<code>text</code>		Valor máximo permitido para o parâmetro (NULL para valores não numéricos)

A visão `pg_settings` não aceita inserções ou exclusões, mas aceita atualizações. Um comando `UPDATE` aplicado a uma linha de `pg_settings` equivale a executar o comando `SET` no parâmetro com este nome. A mudança somente afeta o valor utilizado pela sessão corrente. Se o comando `UPDATE` for executado dentro de uma transação interrompida posteriormente, o efeito deste comando desaparece quando a transação é desfeita. Se a transação onde o comando se encontra for efetivada, os efeitos do comando persistem até o fim da sessão, a não ser que seja substituído por outro comando `UPDATE` ou `SET`.

### 41.36. `pg_stats`

A visão `pg_stats` fornece acesso a informações armazenadas no catálogo `pg_statistic`. Esta visão permite acessar somente as linhas de `pg_statistic` que correspondem às tabelas que o usuário tem permissão para ler e, portanto, é seguro permitir acesso público de leitura a esta visão.

A visão `pg_stats` também foi projetada para mostrar as informações em uma forma mais facilmente lida do que o catálogo subjacente — ao custo de seu esquema ter que ser estendido toda vez que novos tipos de encaixe forem definidos em `pg_statistic`.

Tabela 41-36. Colunas de `pg_stats`

Nome	Tipo	Referencia	Descrição
<code>schemaname</code>	<code>name</code>	<code>pg_namespace.nspname</code>	Nome do esquema que contém a tabela
<code>tablename</code>	<code>name</code>	<code>pg_class.relname</code>	Nome da tabela
<code>attname</code>	<code>name</code>	<code>pg_attribute.attname</code>	Nome da coluna descrita por esta linha
<code>null_frac</code>	<code>real</code>		Fração das entradas nulas na coluna
<code>avg_width</code>	<code>integer</code>		Largura média em bytes das entradas da coluna
<code>n_distinct</code>	<code>real</code>		Se for maior que zero, o número estimado de valores distintos na coluna. Se for menor que zero, o negativo do número de valores distintos divididos pelo número de linhas (A forma negativa é utilizada quando o <code>ANALYZE</code> acredita que o número de valores distintos deverá aumentar quando a tabela crescer; a forma positiva é utilizada quando a coluna parece ter um número fixo de valores possíveis). Por exemplo, -1 indica uma coluna com restrição de unicidade, onde o

Nome	Tipo	Referencia	Descrição
			número de valores distintos é igual ao número de linhas.
most_common_vals	anyarray		Lista dos valores mais comuns da coluna (NULL se nenhum valor parecer ser mais comum que os outros)
most_common_freqs	real[]		Lista das frequências dos valores mais comuns, ou seja, o número de ocorrências de cada um deles dividido pelo número total de linhas. (NULL quando most_common_vals também é nulo).
histogram_bounds	anyarray		Lista dos valores que dividem os valores das colunas em grupos com populações aproximadamente iguais. Os valores em most_common_vals, se estiverem presentes, são omitidos no cálculo deste histograma (Esta coluna é NULL se o tipo de dado da coluna não possuir o operador <, ou se a lista de most_common_vals englobar toda as linhas da tabela).
correlation	real		Correlação estatística entre a ordenação física das linhas e a ordenação lógica dos valores da coluna. Varia de -1 a +1. Quando o valor estiver próximo de -1 ou de +1, uma varredura de índice na coluna será estimada como mais barata do que quando estiver próximo de zero, por causa da redução de acesso randômico ao disco (O valor desta coluna é NULL quando o tipo de dado da coluna não possui um operador <)

O número máximo de entradas nas matrizes `most_common_vals` e `histogram_bounds` podem ser definidos coluna por coluna utilizando o comando `ALTER TABLE SET STATISTICS`, ou globalmente definindo o parâmetro em tempo de execução `default_statistics_target`.

## 41.37. pg\_tables

A visão `pg_tables` fornece acesso a informações úteis sobre todas as tabelas do banco de dados.

**Tabela 41-37. Colunas de `pg_tables`**

Nome	Tipo	Referencia	Descrição
schemaname	name	<code>pg_namespace.nspname</code>	Nome do esquema que contém a tabela
tablename	name	<code>pg_class.relname</code>	Nome da tabela
tableowner	name	<code>pg_shadow.username</code>	Nome do dono da tabela
tablespace	name	<code>pg_tablespace.spcname</code>	Nome do espaço de tabelas que contém a tabela (NULL se for o padrão para o banco de dados)
hasindexes	boolean	<code>pg_class.relhasindex</code>	Verdade se a tabela possui (ou possuía recentemente) algum índice
hasrules	boolean	<code>pg_class.relhasrules</code>	Verdade se a tabela possui regras
hastriggers	boolean	<code>pg_class.reltriggers</code>	Verdade se a tabela possui gatilhos

### 41.38. pg\_user

A visão `pg_user` fornece acesso a informações sobre usuários do banco de dados. É simplesmente uma visão de `pg_shadow` que pode ser lida por todos porque esconde o campo senha.

**Tabela 41-38. Colunas de `pg_user`**

Nome	Tipo	Referencia	Descrição
<code>username</code>	<code>name</code>		Nome do usuário
<code>usesysid</code>	<code>int4</code>		Id do usuário (número arbitrário utilizado para fazer referência a este usuário)
<code>usecreatedb</code>	<code>bool</code>		Se for verdade o usuário pode criar bancos de dados
<code>usesuper</code>	<code>bool</code>		Se for verdade o usuário é um superusuário
<code>usecatupd</code>	<code>bool</code>		Se for verdade o usuário pode atualizar os catálogos do sistema (Mesmo os superusuários não podem atualizar os catálogos do sistema a menos que o valor desta coluna seja verdade).
<code>passwd</code>	<code>text</code>		Não é a senha (sempre mostrado como <code>*****</code> )
<code>valuntil</code>	<code>abstime</code>		Momento de expiração da conta (utilizado apenas para autenticação da senha)
<code>useconfig</code>	<code>text[]</code>		Padrões da sessão para as variáveis de configuração em tempo de execução

### 41.39. pg\_views

A visão `pg_views` fornece acesso a informações úteis sobre todas as visões do banco de dados.

**Tabela 41-39. Colunas de `pg_views`**

Nome	Tipo	Referencia	Descrição
<code>schemaname</code>	<code>name</code>	<code>pg_namespace.nspname</code>	Nome do esquema que contém a visão
<code>viewname</code>	<code>name</code>	<code>pg_class.relname</code>	Nome da visão
<code>viewowner</code>	<code>name</code>	<code>pg_shadow.username</code>	Nome do dono da visão
<code>definition</code>	<code>text</code>		Definição da visão (o comando <code>SELECT</code> reconstruído)

## Notas

1. No diretório do tutorial existe o arquivo `syscat.sql` (`./syscat.sql`) contendo várias consultas interessantes aos catálogos do sistema. (N. do T.)

# Capítulo 42. Frontend/Backend Protocol

PostgreSQL uses a message-based protocol for communication between frontends and backends (clients and servers). The protocol is supported over TCP/IP and also over Unix-domain sockets. Port number 5432 has been registered with IANA as the customary TCP port number for servers supporting this protocol, but in practice any non-privileged port number may be used.

This document describes version 3.0 of the protocol, implemented in PostgreSQL 7.4 and later. For descriptions of the earlier protocol versions, see previous releases of the PostgreSQL documentation. A single server can support multiple protocol versions. The initial startup-request message tells the server which protocol version the client is attempting to use, and then the server follows that protocol if it is able.

Higher level features built on this protocol (for example, how libpq passes certain environment variables when the connection is established) are covered elsewhere.

In order to serve multiple clients efficiently, the server launches a new “backend” process for each client. In the current implementation, a new child process is created immediately after an incoming connection is detected. This is transparent to the protocol, however. For purposes of the protocol, the terms “backend” and “server” are interchangeable; likewise “frontend” and “client” are interchangeable.

## 42.1. Visão geral

The protocol has separate phases for startup and normal operation. In the startup phase, the frontend opens a connection to the server and authenticates itself to the satisfaction of the server. (This might involve a single message, or multiple messages depending on the authentication method being used.) If all goes well, the server then sends status information to the frontend, and finally enters normal operation. Except for the initial startup-request message, this part of the protocol is driven by the server.

During normal operation, the frontend sends queries and other commands to the backend, and the backend sends back query results and other responses. There are a few cases (such as `NOTIFY`) wherein the backend will send unsolicited messages, but for the most part this portion of a session is driven by frontend requests.

Termination of the session is normally by frontend choice, but can be forced by the backend in certain cases. In any case, when the backend closes the connection, it will roll back any open (incomplete) transaction before exiting.

Within normal operation, SQL commands can be executed through either of two sub-protocols. In the “simple query” protocol, the frontend just sends a textual query string, which is parsed and immediately executed by the backend. In the “extended query” protocol, processing of queries is separated into multiple steps: parsing, binding of parameter values, and execution. This offers flexibility and performance benefits, at the cost of extra complexity.

Normal operation has additional sub-protocols for special operations such as `COPY`.

### 42.1.1. Messaging Overview

All communication is through a stream of messages. The first byte of a message identifies the message type, and the next four bytes specify the length of the rest of the message (this length count includes itself, but not the message-type byte). The remaining contents of the message are determined by the message type. For historical reasons, the very first message sent by the client (the startup message) has no initial message-type byte.

To avoid losing synchronization with the message stream, both servers and clients typically read an entire message into a buffer (using the byte count) before attempting to process its contents. This allows easy recovery if an error is detected while processing the contents. In extreme situations (such as not having enough memory to buffer the message), the receiver may use the byte count to determine how much input to skip before it resumes reading messages.

Conversely, both servers and clients must take care never to send an incomplete message. This is commonly done by marshaling the entire message in a buffer before beginning to send it. If a communications failure occurs partway through sending or receiving a message, the only sensible response is to abandon the connection, since there is little hope of recovering message-boundary synchronization.

### 42.1.2. Extended Query Overview

In the extended-query protocol, execution of SQL commands is divided into multiple steps. The state retained between steps is represented by two types of objects: *prepared statements* and *portals*. A prepared statement represents the result of parsing, semantic analysis, and planning of a textual query string. A prepared statement is not necessarily ready to execute, because it may lack specific values for *parameters*. A portal represents a ready-to-execute or already-partially-executed statement, with any missing parameter values filled in. (For `SELECT` statements, a portal is equivalent to an open cursor, but we choose to use a different term since cursors don't handle non-`SELECT` statements.)

The overall execution cycle consists of a *parse* step, which creates a prepared statement from a textual query string; a *bind* step, which creates a portal given a prepared statement and values for any needed parameters; and an *execute* step that runs a portal's query. In the case of a query that returns rows (`SELECT`, `SHOW`, etc), the execute step can be told to fetch only a limited number of rows, so that multiple execute steps may be needed to complete the operation.

The backend can keep track of multiple prepared statements and portals (but note that these exist only within a session, and are never shared across sessions). Existing prepared statements and portals are referenced by names assigned when they were created. In addition, an “unnamed” prepared statement and portal exist. Although these behave largely the same as named objects, operations on them are optimized for the case of executing a query only once and then discarding it, whereas operations on named objects are optimized on the expectation of multiple uses.

### 42.1.3. Formats and Format Codes

Data of a particular data type might be transmitted in any of several different *formats*. As of PostgreSQL 7.4 the only supported formats are “text” and “binary”, but the protocol makes provision for future extensions. The desired format for any value is specified by a *format code*. Clients may specify a format code for each transmitted parameter value and for each column of a query result. Text has format code zero, binary has format code one, and all other format codes are reserved for future definition.

The text representation of values is whatever strings are produced and accepted by the input/output conversion functions for the particular data type. In the transmitted representation, there is no trailing null character; the frontend must add one to received values if it wants to process them as C strings. (The text format does not allow embedded nulls, by the way.)

Binary representations for integers use network byte order (most significant byte first). For other data types consult the documentation or source code to learn about the binary representation. Keep in mind that binary representations for complex data types may change across server versions; the text format is usually the more portable choice.

## 42.2. Message Flow

This section describes the message flow and the semantics of each message type. (Details of the exact representation of each message appear in Seção 42.4.) There are several different sub-protocols depending on the state of the connection: start-up, query, function call, `COPY`, and termination. There are also special provisions for asynchronous operations (including notification responses and command cancellation), which can occur at any time after the start-up phase.

### 42.2.1. Start-Up

To begin a session, a frontend opens a connection to the server and sends a startup message. This message includes the names of the user and of the database the user wants to connect to; it also identifies the particular protocol version to be used. (Optionally, the startup message can include additional settings for run-time parameters.) The server then uses this information and the contents of its configuration files (such as `pg_hba.conf`) to determine whether the connection is provisionally acceptable, and what additional authentication is required (if any).

The server then sends an appropriate authentication request message, to which the frontend must reply with an appropriate authentication response message (such as a password). In principle the authentication request/response cycle could require multiple iterations, but none of the present authentication methods use more than one request and response. In some methods, no response at all is needed from the frontend, and so no authentication request occurs.

The authentication cycle ends with the server either rejecting the connection attempt (`ErrorResponse`), or sending `AuthenticationOk`.

The possible messages from the server in this phase are:

**ErrorResponse**

The connection attempt has been rejected. The server then immediately closes the connection.

**AuthenticationOk**

The authentication exchange is successfully completed.

**AuthenticationKerberosV4**

The frontend must now take part in a Kerberos V4 authentication dialog (not described here, part of the Kerberos specification) with the server. If this is successful, the server responds with an **AuthenticationOk**, otherwise it responds with an **ErrorResponse**.

**AuthenticationKerberosV5**

The frontend must now take part in a Kerberos V5 authentication dialog (not described here, part of the Kerberos specification) with the server. If this is successful, the server responds with an **AuthenticationOk**, otherwise it responds with an **ErrorResponse**.

**AuthenticationCleartextPassword**

The frontend must now send a **PasswordMessage** containing the password in clear-text form. If this is the correct password, the server responds with an **AuthenticationOk**, otherwise it responds with an **ErrorResponse**.

**AuthenticationCryptPassword**

The frontend must now send a **PasswordMessage** containing the password encrypted via `crypt(3)`, using the 2-character salt specified in the **AuthenticationCryptPassword** message. If this is the correct password, the server responds with an **AuthenticationOk**, otherwise it responds with an **ErrorResponse**.

**AuthenticationMD5Password**

The frontend must now send a **PasswordMessage** containing the password encrypted via MD5, using the 4-character salt specified in the **AuthenticationMD5Password** message. If this is the correct password, the server responds with an **AuthenticationOk**, otherwise it responds with an **ErrorResponse**.

**AuthenticationSCMCredential**

This response is only possible for local Unix-domain connections on platforms that support SCM credential messages. The frontend must issue an SCM credential message and then send a single data byte. (The contents of the data byte are uninteresting; it's only used to ensure that the server waits long enough to receive the credential message.) If the credential is acceptable, the server responds with an **AuthenticationOk**, otherwise it responds with an **ErrorResponse**.

If the frontend does not support the authentication method requested by the server, then it should immediately close the connection.

After having received **AuthenticationOk**, the frontend must wait for further messages from the server. In this phase a backend process is being started, and the frontend is just an interested bystander. It is still possible for the startup attempt to fail (**ErrorResponse**), but in the normal case the backend will send some **ParameterStatus** messages, **BackendKeyData**, and finally **ReadyForQuery**.

During this phase the backend will attempt to apply any additional run-time parameter settings that were given in the startup message. If successful, these values become session defaults. An error causes **ErrorResponse** and exit.

The possible messages from the backend in this phase are:

**BackendKeyData**

This message provides secret-key data that the frontend must save if it wants to be able to issue cancel requests later. The frontend should not respond to this message, but should continue listening for a **ReadyForQuery** message.

**ParameterStatus**

This message informs the frontend about the current (initial) setting of backend parameters, such as `client_encoding` or `DateStyle`. The frontend may ignore this message, or record the settings for its future use; see [Seção 42.2.6](#) for more details. The frontend should not respond to this message, but should continue listening for a **ReadyForQuery** message.

**ReadyForQuery**

Start-up is completed. The frontend may now issue commands.

**ErrorResponse**

Start-up failed. The connection is closed after sending this message.

**NoticeResponse**

A warning message has been issued. The frontend should display the message but continue listening for ReadyForQuery or ErrorResponse.

The ReadyForQuery message is the same one that the backend will issue after each command cycle. Depending on the coding needs of the frontend, it is reasonable to consider ReadyForQuery as starting a command cycle, or to consider ReadyForQuery as ending the start-up phase and each subsequent command cycle.

**42.2.2. Simple Query**

A simple query cycle is initiated by the frontend sending a Query message to the backend. The message includes an SQL command (or commands) expressed as a text string. The backend then sends one or more response messages depending on the contents of the query command string, and finally a ReadyForQuery response message. ReadyForQuery informs the frontend that it may safely send a new command. (It is not actually necessary for the frontend to wait for ReadyForQuery before issuing another command, but the frontend must then take responsibility for figuring out what happens if the earlier command fails and already-issued later commands succeed.)

The possible response messages from the backend are:

**CommandComplete**

An SQL command completed normally.

**CopyInResponse**

The backend is ready to copy data from the frontend to a table; see Seção 42.2.5.

**CopyOutResponse**

The backend is ready to copy data from a table to the frontend; see Seção 42.2.5.

**RowDescription**

Indicates that rows are about to be returned in response to a `SELECT`, `FETCH`, etc query. The contents of this message describe the column layout of the rows. This will be followed by a DataRow message for each row being returned to the frontend.

**DataRow**

One of the set of rows returned by a `SELECT`, `FETCH`, etc query.

**EmptyQueryResponse**

An empty query string was recognized.

**ErrorResponse**

An error has occurred.

**ReadyForQuery**

Processing of the query string is complete. A separate message is sent to indicate this because the query string may contain multiple SQL commands. (CommandComplete marks the end of processing one SQL command, not the whole string.) ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

**NoticeResponse**

A warning message has been issued in relation to the query. Notices are in addition to other responses, i.e., the backend will continue processing the command.



The response to a `SELECT` query (or other queries that return row sets, such as `EXPLAIN` or `SHOW`) normally consists of `RowDescription`, zero or more `DataRow` messages, and then `CommandComplete`. `COPY` to or from the frontend invokes special protocol as described in Seção 42.2.5. All other query types normally produce only a `CommandComplete` message.

Since a query string could contain several queries (separated by semicolons), there might be several such response sequences before the backend finishes processing the query string. `ReadyForQuery` is issued when the entire string has been processed and the backend is ready to accept a new query string.

If a completely empty (no contents other than whitespace) query string is received, the response is `EmptyQueryResponse` followed by `ReadyForQuery`.

In the event of an error, `ErrorResponse` is issued followed by `ReadyForQuery`. All further processing of the query string is aborted by `ErrorResponse` (even if more queries remained in it). Note that this may occur partway through the sequence of messages generated by an individual query.

In simple Query mode, the format of retrieved values is always text, except when the given command is a `FETCH` from a cursor declared with the `BINARY` option. In that case, the retrieved values are in binary format. The format codes given in the `RowDescription` message tell which format is being used.

A frontend must be prepared to accept `ErrorResponse` and `NoticeResponse` messages whenever it is expecting any other type of message. See also Seção 42.2.6 concerning messages that the backend may generate due to outside events.

Recommended practice is to code frontends in a state-machine style that will accept any message type at any time that it could make sense, rather than wiring in assumptions about the exact sequence of messages.

### 42.2.3. Extended Query

The extended query protocol breaks down the above-described simple query protocol into multiple steps. The results of preparatory steps can be re-used multiple times for improved efficiency. Furthermore, additional features are available, such as the possibility of supplying data values as separate parameters instead of having to insert them directly into a query string.

In the extended protocol, the frontend first sends a `Parse` message, which contains a textual query string, optionally some information about data types of parameter placeholders, and the name of a destination prepared-statement object (an empty string selects the unnamed prepared statement). The response is either `ParseComplete` or `ErrorResponse`. Parameter data types may be specified by `OID`; if not given, the parser attempts to infer the data types in the same way as it would do for untyped literal string constants.

**Nota:** The query string contained in a `Parse` message cannot include more than one SQL statement; else a syntax error is reported. This restriction does not exist in the simple-query protocol, but it does exist in the extended protocol, because allowing prepared statements or portals to contain multiple commands would complicate the protocol unduly.

If successfully created, a named prepared-statement object lasts till the end of the current session, unless explicitly destroyed. An unnamed prepared statement lasts only until the next `Parse` statement specifying the unnamed statement as destination is issued. (Note that a simple Query message also destroys the unnamed statement.) Named prepared statements must be explicitly closed before they can be redefined by a `Parse` message, but this is not required for the unnamed statement. Named prepared statements can also be created and accessed at the SQL command level, using `PREPARE` and `EXECUTE`.

Once a prepared statement exists, it can be readied for execution using a `Bind` message. The `Bind` message gives the name of the source prepared statement (empty string denotes the unnamed prepared statement), the name of the destination portal (empty string denotes the unnamed portal), and the values to use for any parameter placeholders present in the prepared statement. The supplied parameter set must match those needed by the prepared statement. `Bind` also specifies the format to use for any data returned by the query; the format can be specified overall, or per-column. The response is either `BindComplete` or `ErrorResponse`.

**Nota:** The choice between text and binary output is determined by the format codes given in `Bind`, regardless of the SQL command involved. The `BINARY` attribute in cursor declarations is irrelevant when using extended query protocol.

Query planning for named prepared-statement objects occurs when the `Parse` message is received. If a query will be repeatedly executed with different parameters, it may be beneficial to send a single `Parse` message containing a

parameterized query, followed by multiple Bind and Execute messages. This will avoid replanning the query on each execution.

The unnamed prepared statement is likewise planned during Parse processing if the Parse message defines no parameters. But if there are parameters, query planning is delayed until the first Bind message for the statement is received. The planner will consider the actual values of the parameters provided in the Bind message when planning the query.

**Nota:** Query plans generated from a parameterized query may be less efficient than query plans generated from an equivalent query with actual parameter values substituted. The query planner cannot make decisions based on actual parameter values (for example, index selectivity) when planning a parameterized query assigned to a named prepared-statement object. This possible penalty is avoided when using the unnamed statement, since it is not planned until actual parameter values are available.

If a second or subsequent Bind referencing the unnamed prepared-statement object is received without an intervening Parse, the query is not replanned. The parameter values used in the first Bind message may produce a query plan that is only efficient for a subset of possible parameter values. To force replanning of the query for a fresh set of parameters, send another Parse message to replace the unnamed prepared-statement object.

If successfully created, a named portal object lasts till the end of the current transaction, unless explicitly destroyed. An unnamed portal is destroyed at the end of the transaction, or as soon as the next Bind statement specifying the unnamed portal as destination is issued. (Note that a simple Query message also destroys the unnamed portal.) Named portals must be explicitly closed before they can be redefined by a Bind message, but this is not required for the unnamed portal. Named portals can also be created and accessed at the SQL command level, using `DECLARE CURSOR` and `FETCH`.

Once a portal exists, it can be executed using an Execute message. The Execute message specifies the portal name (empty string denotes the unnamed portal) and a maximum result-row count (zero meaning “fetch all rows”). The result-row count is only meaningful for portals containing commands that return row sets; in other cases the command is always executed to completion, and the row count is ignored. The possible responses to Execute are the same as those described above for queries issued via simple query protocol, except that Execute doesn't cause ReadyForQuery or RowDescription to be issued.

If Execute terminates before completing the execution of a portal (due to reaching a nonzero result-row count), it will send a PortalSuspended message; the appearance of this message tells the frontend that another Execute should be issued against the same portal to complete the operation. The CommandComplete message indicating completion of the source SQL command is not sent until the portal's execution is completed. Therefore, an Execute phase is always terminated by the appearance of exactly one of these messages: CommandComplete, EmptyQueryResponse (if the portal was created from an empty query string), ErrorResponse, or PortalSuspended.

At completion of each series of extended-query messages, the frontend should issue a Sync message. This parameterless message causes the backend to close the current transaction if it's not inside a `BEGIN/COMMIT` transaction block (“close” meaning to commit if no error, or roll back if error). Then a ReadyForQuery response is issued. The purpose of Sync is to provide a resynchronization point for error recovery. When an error is detected while processing any extended-query message, the backend issues ErrorResponse, then reads and discards messages until a Sync is reached, then issues ReadyForQuery and returns to normal message processing. (But note that no skipping occurs if an error is detected *while* processing Sync — this ensures that there is one and only one ReadyForQuery sent for each Sync.)

**Nota:** Sync does not cause a transaction block opened with `BEGIN` to be closed. It is possible to detect this situation since the ReadyForQuery message includes transaction status information.

In addition to these fundamental, required operations, there are several optional operations that can be used with extended-query protocol.

The Describe message (portal variant) specifies the name of an existing portal (or an empty string for the unnamed portal). The response is a RowDescription message describing the rows that will be returned by executing the portal; or a NoData message if the portal does not contain a query that will return rows; or ErrorResponse if there is no such portal.

The Describe message (statement variant) specifies the name of an existing prepared statement (or an empty string for the unnamed prepared statement). The response is a ParameterDescription message describing the parameters needed by the statement, followed by a RowDescription message describing the rows that will be returned when the statement is eventually executed (or a NoData message if the statement will not return rows). ErrorResponse is issued if there is no such prepared statement. Note that since Bind has not yet been issued, the formats to be used for returned columns are not yet known to the backend; the format code fields in the RowDescription message will be zeroes in this case.

**Dica:** In most scenarios the frontend should issue one or the other variant of Describe before issuing Execute, to ensure that it knows how to interpret the results it will get back.

The Close message closes an existing prepared statement or portal and releases resources. It is not an error to issue Close against a nonexistent statement or portal name. The response is normally CloseComplete, but could be ErrorResponse if some difficulty is encountered while releasing resources. Note that closing a prepared statement implicitly closes any open portals that were constructed from that statement.

The Flush message does not cause any specific output to be generated, but forces the backend to deliver any data pending in its output buffers. A Flush must be sent after any extended-query command except Sync, if the frontend wishes to examine the results of that command before issuing more commands. Without Flush, messages returned by the backend will be combined into the minimum possible number of packets to minimize network overhead.

**Nota:** The simple Query message is approximately equivalent to the series Parse, Bind, portal Describe, Execute, Close, Sync, using the unnamed prepared statement and portal objects and no parameters. One difference is that it will accept multiple SQL statements in the query string, automatically performing the bind/describe/execute sequence for each one in succession. Another difference is that it will not return ParseComplete, BindComplete, CloseComplete, or NoData messages.

#### 42.2.4. Function Call

The Function Call sub-protocol allows the client to request a direct call of any function that exists in the database's `pg_proc` system catalog. The client must have execute permission for the function.

**Nota:** The Function Call sub-protocol is a legacy feature that is probably best avoided in new code. Similar results can be accomplished by setting up a prepared statement that does `SELECT function($1, ...)`. The Function Call cycle can then be replaced with Bind/Execute.

A Function Call cycle is initiated by the frontend sending a FunctionCall message to the backend. The backend then sends one or more response messages depending on the results of the function call, and finally a ReadyForQuery response message. ReadyForQuery informs the frontend that it may safely send a new query or function call.

The possible response messages from the backend are:

ErrorResponse

An error has occurred.

FunctionCallResponse

The function call was completed and returned the result given in the message. (Note that the Function Call protocol can only handle a single scalar result, not a row type or set of results.)

ReadyForQuery

Processing of the function call is complete. ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

NoticeResponse

A warning message has been issued in relation to the function call. Notices are in addition to other responses, i.e., the backend will continue processing the command.

#### 42.2.5. COPY Operations

The COPY command allows high-speed bulk data transfer to or from the server. Copy-in and copy-out operations each switch the connection into a distinct sub-protocol, which lasts until the operation is completed.

Copy-in mode (data transfer to the server) is initiated when the backend executes a `COPY FROM STDIN` SQL statement. The backend sends a CopyInResponse message to the frontend. The frontend should then send zero or more CopyData messages, forming a stream of input data. (The message boundaries are not required to have anything to do with row boundaries, although that is often a reasonable choice.) The frontend can terminate the copy-in mode by sending either a CopyDone message (allowing successful termination) or a CopyFail message (which will cause the COPY SQL statement to fail with an error). The backend then reverts to the command-processing mode it was in before the COPY started, which will

be either simple or extended query protocol. It will next send either `CommandComplete` (if successful) or `ErrorResponse` (if not).

In the event of a backend-detected error during copy-in mode (including receipt of a `CopyFail` message), the backend will issue an `ErrorResponse` message. If the `COPY` command was issued via an extended-query message, the backend will now discard frontend messages until a `Sync` message is received, then it will issue `ReadyForQuery` and return to normal processing. If the `COPY` command was issued in a simple `Query` message, the rest of that message is discarded and `ReadyForQuery` is issued. In either case, any subsequent `CopyData`, `CopyDone`, or `CopyFail` messages issued by the frontend will simply be dropped.

The backend will ignore `Flush` and `Sync` messages received during copy-in mode. Receipt of any other non-copy message type constitutes an error that will abort the copy-in state as described above. (The exception for `Flush` and `Sync` is for the convenience of client libraries that always send `Flush` or `Sync` after an `Execute` message, without checking whether the command to be executed is a `COPY FROM STDIN`.)

Copy-out mode (data transfer from the server) is initiated when the backend executes a `COPY TO STDOUT SQL` statement. The backend sends a `CopyOutResponse` message to the frontend, followed by zero or more `CopyData` messages (always one per row), followed by `CopyDone`. The backend then reverts to the command-processing mode it was in before the `COPY` started, and sends `CommandComplete`. The frontend cannot abort the transfer (except by closing the connection or issuing a `Cancel` request), but it can discard unwanted `CopyData` and `CopyDone` messages.

In the event of a backend-detected error during copy-out mode, the backend will issue an `ErrorResponse` message and revert to normal processing. The frontend should treat receipt of `ErrorResponse` (or indeed any message type other than `CopyData` or `CopyDone`) as terminating the copy-out mode.

The `CopyInResponse` and `CopyOutResponse` messages include fields that inform the frontend of the number of columns per row and the format codes being used for each column. (As of the present implementation, all columns in a given `COPY` operation will use the same format, but the message design does not assume this.)

#### 42.2.6. Asynchronous Operations

There are several cases in which the backend will send messages that are not specifically prompted by the frontend's command stream. Frontends must be prepared to deal with these messages at any time, even when not engaged in a query. At minimum, one should check for these cases before beginning to read a query response.

It is possible for `NoticeResponse` messages to be generated due to outside activity; for example, if the database administrator commands a “fast” database shutdown, the backend will send a `NoticeResponse` indicating this fact before closing the connection. Accordingly, frontends should always be prepared to accept and display `NoticeResponse` messages, even when the connection is nominally idle.

`ParameterStatus` messages will be generated whenever the active value changes for any of the parameters the backend believes the frontend should know about. Most commonly this occurs in response to a `SET SQL` command executed by the frontend, and this case is effectively synchronous — but it is also possible for parameter status changes to occur because the administrator changed a configuration file and then sent the `SIGHUP` signal to the postmaster. Also, if a `SET` command is rolled back, an appropriate `ParameterStatus` message will be generated to report the current effective value.

At present there is a hard-wired set of parameters for which `ParameterStatus` will be generated: they are `server_version`, `server_encoding`, `client_encoding`, `is_superuser`, `session_authorization`, `DateStyle`, `TimeZone`, and `integer_datetimes`. (`server_encoding`, `TimeZone`, and `integer_datetimes` were not reported by releases before 8.0.) Note that `server_version`, `server_encoding` and `integer_datetimes` are pseudo-parameters that cannot change after startup. This set might change in the future, or even become configurable. Accordingly, a frontend should simply ignore `ParameterStatus` for parameters that it does not understand or care about.

If a frontend issues a `LISTEN` command, then the backend will send a `NotificationResponse` message (not to be confused with `NoticeResponse`!) whenever a `NOTIFY` command is executed for the same notification name.

**Nota:** At present, `NotificationResponse` can only be sent outside a transaction, and thus it will not occur in the middle of a command-response series, though it may occur just before `ReadyForQuery`. It is unwise to design frontend logic that assumes that, however. Good practice is to be able to accept `NotificationResponse` at any point in the protocol.

### 42.2.7. Cancelling Requests in Progress

During the processing of a query, the frontend may request cancellation of the query. The cancel request is not sent directly on the open connection to the backend for reasons of implementation efficiency: we don't want to have the backend constantly checking for new input from the frontend during query processing. Cancel requests should be relatively infrequent, so we make them slightly cumbersome in order to avoid a penalty in the normal case.

To issue a cancel request, the frontend opens a new connection to the server and sends a `CancelRequest` message, rather than the `StartupMessage` message that would ordinarily be sent across a new connection. The server will process this request and then close the connection. For security reasons, no direct reply is made to the cancel request message.

A `CancelRequest` message will be ignored unless it contains the same key data (PID and secret key) passed to the frontend during connection start-up. If the request matches the PID and secret key for a currently executing backend, the processing of the current query is aborted. (In the existing implementation, this is done by sending a special signal to the backend process that is processing the query.)

The cancellation signal may or may not have any effect — for example, if it arrives after the backend has finished processing the query, then it will have no effect. If the cancellation is effective, it results in the current command being terminated early with an error message.

The upshot of all this is that for reasons of both security and efficiency, the frontend has no direct way to tell whether a cancel request has succeeded. It must continue to wait for the backend to respond to the query. Issuing a cancel simply improves the odds that the current query will finish soon, and improves the odds that it will fail with an error message instead of succeeding.

Since the cancel request is sent across a new connection to the server and not across the regular frontend/backend communication link, it is possible for the cancel request to be issued by any process, not just the frontend whose query is to be canceled. This may have some benefits of flexibility in building multiple-process applications. It also introduces a security risk, in that unauthorized persons might try to cancel queries. The security risk is addressed by requiring a dynamically generated secret key to be supplied in cancel requests.

### 42.2.8. Termination

The normal, graceful termination procedure is that the frontend sends a `Terminate` message and immediately closes the connection. On receipt of this message, the backend closes the connection and terminates.

In rare cases (such as an administrator-commanded database shutdown) the backend may disconnect without any frontend request to do so. In such cases the backend will attempt to send an error or notice message giving the reason for the disconnection before it closes the connection.

Other termination scenarios arise from various failure cases, such as core dump at one end or the other, loss of the communications link, loss of message-boundary synchronization, etc. If either frontend or backend sees an unexpected closure of the connection, it should clean up and terminate. The frontend has the option of launching a new backend by recontacting the server if it doesn't want to terminate itself. Closing the connection is also advisable if an unrecognizable message type is received, since this probably indicates loss of message-boundary sync.

For either normal or abnormal termination, any open transaction is rolled back, not committed. One should note however that if a frontend disconnects while a non-`SELECT` query is being processed, the backend will probably finish the query before noticing the disconnection. If the query is outside any transaction block (`BEGIN ... COMMIT` sequence) then its results may be committed before the disconnection is recognized.

### 42.2.9. SSL Session Encryption

If PostgreSQL was built with SSL support, frontend/backend communications can be encrypted using SSL. This provides communication security in environments where attackers might be able to capture the session traffic. For more information on encrypting PostgreSQL sessions with SSL, see Seção 16.7.

To initiate an SSL-encrypted connection, the frontend initially sends an `SSLRequest` message rather than a `StartupMessage`. The server then responds with a single byte containing `S` or `N`, indicating that it is willing or unwilling to perform SSL, respectively. The frontend may close the connection at this point if it is dissatisfied with the response. To continue after `S`, perform an SSL startup handshake (not described here, part of the SSL specification) with the server. If this is successful, continue with sending the usual `StartupMessage`. In this case the `StartupMessage` and all subsequent data will be SSL-encrypted. To continue after `N`, send the usual `StartupMessage` and proceed without encryption.

The frontend should also be prepared to handle an ErrorMessage response to SSLRequest from the server. This would only occur if the server predates the addition of SSL support to PostgreSQL. In this case the connection must be closed, but the frontend may choose to open a fresh connection and proceed without requesting SSL.

An initial SSLRequest may also be used in a connection that is being opened to send a CancelRequest message.

While the protocol itself does not provide a way for the server to force SSL encryption, the administrator may configure the server to reject unencrypted sessions as a byproduct of authentication checking.

### 42.3. Message Data Types

This section describes the base data types used in messages.

`Intn(i)`

An  $n$ -bit integer in network byte order (most significant byte first). If  $i$  is specified it is the exact value that will appear, otherwise the value is variable. Eg. `Int16`, `Int32(42)`.

`Intn[k]`

An array of  $k$   $n$ -bit integers, each in network byte order. The array length  $k$  is always determined by an earlier field in the message. Eg. `Int16[M]`.

`String(s)`

A null-terminated string (C-style string). There is no specific length limitation on strings. If  $s$  is specified it is the exact value that will appear, otherwise the value is variable. Eg. `String`, `String("user")`.

**Nota:** *There is no predefined limit on the length of a string that can be returned by the backend. Good coding strategy for a frontend is to use an expandable buffer so that anything that fits in memory can be accepted. If that's not feasible, read the full string and discard trailing characters that don't fit into your fixed-size buffer.*

`Byten(c)`

Exactly  $n$  bytes. If the field width  $n$  is not a constant, it is always determinable from an earlier field in the message. If  $c$  is specified it is the exact value. Eg. `Byte2`, `Byte1("\n")`.

### 42.4. Message Formats

This section describes the detailed format of each message. Each is marked to indicate that it may be sent by a frontend (F), a backend (B), or both (F & B). Notice that although each message includes a byte count at the beginning, the message format is defined so that the message end can be found without reference to the byte count. This aids validity checking. (The CopyData message is an exception, because it forms part of a data stream; the contents of any individual CopyData message may not be interpretable on their own.)

AuthenticationOk (B)

`Byte1('R')`

Identifies the message as an authentication request.

`Int32(8)`

Length of message contents in bytes, including self.

`Int32(0)`

Specifies that the authentication was successful.

AuthenticationKerberosV4 (B)

`Byte1('R')`

Identifies the message as an authentication request.

Int32(8)

Length of message contents in bytes, including self.

Int32(1)

Specifies that Kerberos V4 authentication is required.

AuthenticationKerberosV5 (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(8)

Length of message contents in bytes, including self.

Int32(2)

Specifies that Kerberos V5 authentication is required.

AuthenticationCleartextPassword (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(8)

Length of message contents in bytes, including self.

Int32(3)

Specifies that a clear-text password is required.

AuthenticationCryptPassword (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(10)

Length of message contents in bytes, including self.

Int32(4)

Specifies that a crypt()-encrypted password is required.

Byte2

The salt to use when encrypting the password.

AuthenticationMD5Password (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(12)

Length of message contents in bytes, including self.

Int32(5)

Specifies that an MD5-encrypted password is required.

Byte4

The salt to use when encrypting the password.

## AuthenticationSCMCredential (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(8)

Length of message contents in bytes, including self.

Int32(6)

Specifies that an SCM credentials message is required.

## BackendKeyData (B)

Byte1('K')

Identifies the message as cancellation key data. The frontend must save these values if it wishes to be able to issue CancelRequest messages later.

Int32(12)

Length of message contents in bytes, including self.

Int32

The process ID of this backend.

Int32

The secret key of this backend.

## Bind (F)

Byte1('B')

Identifies the message as a Bind command.

Int32

Length of message contents in bytes, including self.

String

The name of the destination portal (an empty string selects the unnamed portal).

String

The name of the source prepared statement (an empty string selects the unnamed prepared statement).

Int16

The number of parameter format codes that follow (denoted *C* below). This can be zero to indicate that there are no parameters or that the parameters all use the default format (text); or one, in which case the specified format code is applied to all parameters; or it can equal the actual number of parameters.Int16[*C*]

The parameter format codes. Each must presently be zero (text) or one (binary).

Int16

The number of parameter values that follow (possibly zero). This must match the number of parameters needed by the query.

Next, the following pair of fields appear for each parameter:

Int32

The length of the parameter value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL parameter value. No value bytes follow in the NULL case.



Byte $n$

The value of the parameter, in the format indicated by the associated format code.  $n$  is the above length.

After the last parameter, the following fields appear:

Int16

The number of result-column format codes that follow (denoted  $R$  below). This can be zero to indicate that there are no result columns or that the result columns should all use the default format (text); or one, in which case the specified format code is applied to all result columns (if any); or it can equal the actual number of result columns of the query.

Int16[ $R$ ]

The result-column format codes. Each must presently be zero (text) or one (binary).

BindComplete (B)

Byte1('2')

Identifies the message as a Bind-complete indicator.

Int32(4)

Length of message contents in bytes, including self.

CancelRequest (F)

Int32(16)

Length of message contents in bytes, including self.

Int32(80877102)

The cancel request code. The value is chosen to contain 1234 in the most significant 16 bits, and 5678 in the least 16 significant bits. (To avoid confusion, this code must not be the same as any protocol version number.)

Int32

The process ID of the target backend.

Int32

The secret key for the target backend.

Close (F)

Byte1('C')

Identifies the message as a Close command.

Int32

Length of message contents in bytes, including self.

Byte1

'S' to close a prepared statement; or 'P' to close a portal.

String

The name of the prepared statement or portal to close (an empty string selects the unnamed prepared statement or portal).

CloseComplete (B)

Byte1('3')

Identifies the message as a Close-complete indicator.

Int32(4)

Length of message contents in bytes, including self.

CommandComplete (B)

Byte1('C')

Identifies the message as a command-completed response.

Int32

Length of message contents in bytes, including self.

String

The command tag. This is usually a single word that identifies which SQL command was completed.

For an INSERT command, the tag is INSERT *oid rows*, where *rows* is the number of rows inserted. *oid* is the object ID of the inserted row if *rows* is 1 and the target table has OIDs; otherwise *oid* is 0.

For a DELETE command, the tag is DELETE *rows* where *rows* is the number of rows deleted.

For an UPDATE command, the tag is UPDATE *rows* where *rows* is the number of rows updated.

For a MOVE command, the tag is MOVE *rows* where *rows* is the number of rows the cursor's position has been changed by.

For a FETCH command, the tag is FETCH *rows* where *rows* is the number of rows that have been retrieved from the cursor.

CopyData (F &amp; B)

Byte1('d')

Identifies the message as COPY data.

Int32

Length of message contents in bytes, including self.

Byten

Data that forms part of a COPY data stream. Messages sent from the backend will always correspond to single data rows, but messages sent by frontends may divide the data stream arbitrarily.

CopyDone (F &amp; B)

Byte1('c')

Identifies the message as a COPY-complete indicator.

Int32(4)

Length of message contents in bytes, including self.

CopyFail (F)

Byte1('f')

Identifies the message as a COPY-failure indicator.

Int32

Length of message contents in bytes, including self.

String

An error message to report as the cause of failure.

## CopyInResponse (B)

Byte1('G')

Identifies the message as a Start Copy In response. The frontend must now send copy-in data (if not prepared to do so, send a CopyFail message).

Int32

Length of message contents in bytes, including self.

Int8

0 indicates the overall COPY format is textual (rows separated by newlines, columns separated by separator characters, etc). 1 indicates the overall copy format is binary (similar to DataRow format). See *COPY* for more information.

Int16

The number of columns in the data to be copied (denoted  $N$  below).

Int16[ $N$ ]

The format codes to be used for each column. Each must presently be zero (text) or one (binary). All must be zero if the overall copy format is textual.

## CopyOutResponse (B)

Byte1('H')

Identifies the message as a Start Copy Out response. This message will be followed by copy-out data.

Int32

Length of message contents in bytes, including self.

Int8

0 indicates the overall COPY format is textual (rows separated by newlines, columns separated by separator characters, etc). 1 indicates the overall copy format is binary (similar to DataRow format). See *COPY* for more information.

Int16

The number of columns in the data to be copied (denoted  $N$  below).

Int16[ $N$ ]

The format codes to be used for each column. Each must presently be zero (text) or one (binary). All must be zero if the overall copy format is textual.

## DataRow (B)

Byte1('D')

Identifies the message as a data row.

Int32

Length of message contents in bytes, including self.

Int16

The number of column values that follow (possibly zero).

Next, the following pair of fields appear for each column:

Int32

The length of the column value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL column value. No value bytes follow in the NULL case.

Byte $n$

The value of the column, in the format indicated by the associated format code.  $n$  is the above length.

Describe (F)

Byte1('D')

Identifies the message as a Describe command.

Int32

Length of message contents in bytes, including self.

Byte1

's' to describe a prepared statement; or 'P' to describe a portal.

String

The name of the prepared statement or portal to describe (an empty string selects the unnamed prepared statement or portal).

EmptyQueryResponse (B)

Byte1('I')

Identifies the message as a response to an empty query string. (This substitutes for CommandComplete.)

Int32(4)

Length of message contents in bytes, including self.

ErrorResponse (B)

Byte1('E')

Identifies the message as an error.

Int32

Length of message contents in bytes, including self.

The message body consists of one or more identified fields, followed by a zero byte as a terminator. Fields may appear in any order. For each field there is the following:

Byte1

A code identifying the field type; if zero, this is the message terminator and no string follows. The presently defined field types are listed in Seção 42.5. Since more field types may be added in future, frontends should silently ignore fields of unrecognized type.

String

The field value.

Execute (F)

Byte1('E')

Identifies the message as an Execute command.

Int32

Length of message contents in bytes, including self.

String

The name of the portal to execute (an empty string selects the unnamed portal).

Int32

Maximum number of rows to return, if portal contains a query that returns rows (ignored otherwise). Zero denotes “no limit”.

Flush (F)

Byte1('H')

Identifies the message as a Flush command.

Int32(4)

Length of message contents in bytes, including self.

FunctionCall (F)

Byte1('F')

Identifies the message as a function call.

Int32

Length of message contents in bytes, including self.

Int32

Specifies the object ID of the function to call.

Int16

The number of argument format codes that follow (denoted *C* below). This can be zero to indicate that there are no arguments or that the arguments all use the default format (text); or one, in which case the specified format code is applied to all arguments; or it can equal the actual number of arguments.

Int16[*C*]

The argument format codes. Each must presently be zero (text) or one (binary).

Int16

Specifies the number of arguments being supplied to the function.

Next, the following pair of fields appear for each argument:

Int32

The length of the argument value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL argument value. No value bytes follow in the NULL case.

Byten

The value of the argument, in the format indicated by the associated format code. *n* is the above length.

After the last argument, the following field appears:

Int16

The format code for the function result. Must presently be zero (text) or one (binary).

FunctionCallResponse (B)

Byte1('V')

Identifies the message as a function call result.

Int32

Length of message contents in bytes, including self.

Int32

The length of the function result value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL function result. No value bytes follow in the NULL case.

Byte $n$

The value of the function result, in the format indicated by the associated format code.  $n$  is the above length.

NoData (B)

Byte1('n')

Identifies the message as a no-data indicator.

Int32(4)

Length of message contents in bytes, including self.

NoticeResponse (B)

Byte1('N')

Identifies the message as a notice.

Int32

Length of message contents in bytes, including self.

The message body consists of one or more identified fields, followed by a zero byte as a terminator. Fields may appear in any order. For each field there is the following:

Byte1

A code identifying the field type; if zero, this is the message terminator and no string follows. The presently defined field types are listed in Seção 42.5. Since more field types may be added in future, frontends should silently ignore fields of unrecognized type.

String

The field value.

NotificationResponse (B)

Byte1('A')

Identifies the message as a notification response.

Int32

Length of message contents in bytes, including self.

Int32

The process ID of the notifying backend process.

String

The name of the condition that the notify has been raised on.

String

Additional information passed from the notifying process. (Currently, this feature is unimplemented so the field is always an empty string.)

ParameterDescription (B)

Byte1('t')

Identifies the message as a parameter description.

Int32

Length of message contents in bytes, including self.

Int16

The number of parameters used by the statement (may be zero).

Then, for each parameter, there is the following:

Int32

Specifies the object ID of the parameter data type.

ParameterStatus (B)

Byte1('S')

Identifies the message as a run-time parameter status report.

Int32

Length of message contents in bytes, including self.

String

The name of the run-time parameter being reported.

String

The current value of the parameter.

Parse (F)

Byte1('P')

Identifies the message as a Parse command.

Int32

Length of message contents in bytes, including self.

String

The name of the destination prepared statement (an empty string selects the unnamed prepared statement).

String

The query string to be parsed.

Int16

The number of parameter data types specified (may be zero). Note that this is not an indication of the number of parameters that might appear in the query string, only the number that the frontend wants to prespecify types for.

Then, for each parameter, there is the following:

Int32

Specifies the object ID of the parameter data type. Placing a zero here is equivalent to leaving the type unspecified.

ParseComplete (B)

Byte1('I')

Identifies the message as a Parse-complete indicator.

Int32(4)

Length of message contents in bytes, including self.

## PasswordMessage (F)

Byte1('p')

Identifies the message as a password response.

Int32

Length of message contents in bytes, including self.

String

The password (encrypted, if requested).

## PortalSuspended (B)

Byte1('s')

Identifies the message as a portal-suspended indicator. Note this only appears if an Execute message's row-count limit was reached.

Int32(4)

Length of message contents in bytes, including self.

## Query (F)

Byte1('Q')

Identifies the message as a simple query.

Int32

Length of message contents in bytes, including self.

String

The query string itself.

## ReadyForQuery (B)

Byte1('Z')

Identifies the message type. ReadyForQuery is sent whenever the backend is ready for a new query cycle.

Int32(5)

Length of message contents in bytes, including self.

Byte1

Current backend transaction status indicator. Possible values are 'I' if idle (not in a transaction block); 'T' if in a transaction block; or 'E' if in a failed transaction block (queries will be rejected until block is ended).

## RowDescription (B)

Byte1('T')

Identifies the message as a row description.

Int32

Length of message contents in bytes, including self.

Int16

Specifies the number of fields in a row (may be zero).

Then, for each field, there is the following:



String

The field name.

Int32

If the field can be identified as a column of a specific table, the object ID of the table; otherwise zero.

Int16

If the field can be identified as a column of a specific table, the attribute number of the column; otherwise zero.

Int32

The object ID of the field's data type.

Int16

The data type size (see `pg_type.typelen`). Note that negative values denote variable-width types.

Int32

The type modifier (see `pg_attribute.atttypmod`). The meaning of the modifier is type-specific.

Int16

The format code being used for the field. Currently will be zero (text) or one (binary). In a `RowDescription` returned from the statement variant of `Describe`, the format code is not yet known and will always be zero.

SSLRequest (F)

Int32(8)

Length of message contents in bytes, including self.

Int32(80877103)

The SSL request code. The value is chosen to contain 1234 in the most significant 16 bits, and 5679 in the least 16 significant bits. (To avoid confusion, this code must not be the same as any protocol version number.)

StartupMessage (F)

Int32

Length of message contents in bytes, including self.

Int32(196608)

The protocol version number. The most significant 16 bits are the major version number (3 for the protocol described here). The least significant 16 bits are the minor version number (0 for the protocol described here).

The protocol version number is followed by one or more pairs of parameter name and value strings. A zero byte is required as a terminator after the last name/value pair. Parameters can appear in any order. `user` is required, others are optional. Each parameter is specified as:

String

The parameter name. Currently recognized names are:

`user`

The database user name to connect as. Required; there is no default.

`database`

The database to connect to. Defaults to the user name.

`options`

Command-line arguments for the backend. (This is deprecated in favor of setting individual run-time parameters.)

In addition to the above, any run-time parameter that can be set at backend start time may be listed. Such settings will be applied during backend start (after parsing the command-line options if any). The values will act as session defaults.

String

The parameter value.

Sync (F)

Byte1('S')

Identifies the message as a Sync command.

Int32(4)

Length of message contents in bytes, including self.

Terminate (F)

Byte1('X')

Identifies the message as a termination.

Int32(4)

Length of message contents in bytes, including self.

## 42.5. Error and Notice Message Fields

This section describes the fields that may appear in ErrorResponse and NoticeResponse messages. Each field type has a single-byte identification token. Note that any given field type should appear at most once per message.

S

Severity: the field contents are `ERROR`, `FATAL`, or `PANIC` (in an error message), or `WARNING`, `NOTICE`, `DEBUG`, `INFO`, or `LOG` (in a notice message), or a localized translation of one of these. Always present.

C

Code: the SQLSTATE code for the error (see Apêndice A). Not localizable. Always present.

M

Message: the primary human-readable error message. This should be accurate but terse (typically one line). Always present.

D

Detail: an optional secondary error message carrying more detail about the problem. May run to multiple lines.

H

Hint: an optional suggestion what to do about the problem. This is intended to differ from Detail in that it offers advice (potentially inappropriate) rather than hard facts. May run to multiple lines.

P

Position: the field value is a decimal ASCII integer, indicating an error cursor position as an index into the original query string. The first character has index 1, and positions are measured in characters not bytes.

P

Internal position: this is defined the same as the P field, but it is used when the cursor position refers to an internally generated command rather than the one submitted by the client. The q field will always appear when this field appears.

q

Internal query: the text of a failed internally-generated command. This could be, for example, a SQL query issued by a PL/pgSQL function.

W

Where: an indication of the context in which the error occurred. Presently this includes a call stack traceback of active procedural language functions and internally-generated queries. The trace is one entry per line, most recent first.

F

File: the file name of the source-code location where the error was reported.

L

Line: the line number of the source-code location where the error was reported.

R

Routine: the name of the source-code routine reporting the error.

The client is responsible for formatting displayed information to meet its needs; in particular it should break long lines as needed. Newline characters appearing in the error message fields should be treated as paragraph breaks, not line breaks.

## 42.6. Summary of Changes since Protocol 2.0

This section provides a quick checklist of changes, for the benefit of developers trying to update existing client libraries to protocol 3.0.

The initial startup packet uses a flexible list-of-strings format instead of a fixed format. Notice that session default values for run-time parameters can now be specified directly in the startup packet. (Actually, you could do that before using the `options` field, but given the limited width of `options` and the lack of any way to quote whitespace in the values, it wasn't a very safe technique.)

All messages now have a length count immediately following the message type byte (except for startup packets, which have no type byte). Also note that `PasswordMessage` now has a type byte.

`ErrorResponse` and `NoticeResponse` ('E' and 'N') messages now contain multiple fields, from which the client code may assemble an error message of the desired level of verbosity. Note that individual fields will typically not end with a newline, whereas the single string sent in the older protocol always did.

The `ReadyForQuery` ('Z') message includes a transaction status indicator.

The distinction between `BinaryRow` and `DataRow` message types is gone; the single `DataRow` message type serves for returning data in all formats. Note that the layout of `DataRow` has changed to make it easier to parse. Also, the representation of binary values has changed: it is no longer directly tied to the server's internal representation.

There is a new “extended query” sub-protocol, which adds the frontend message types `Parse`, `Bind`, `Execute`, `Describe`, `Close`, `Flush`, and `Sync`, and the backend message types `ParseComplete`, `BindComplete`, `PortalSuspended`, `ParameterDescription`, `NoData`, and `CloseComplete`. Existing clients do not have to concern themselves with this sub-protocol, but making use of it may allow improvements in performance or functionality.

`COPY` data is now encapsulated into `CopyData` and `CopyDone` messages. There is a well-defined way to recover from errors during `COPY`. The special “\.” last line is not needed anymore, and is not sent during `COPY OUT`. (It is still recognized as a terminator during `COPY IN`, but its use is deprecated and will eventually be removed.) Binary `COPY` is supported. The `CopyInResponse` and `CopyOutResponse` messages include fields indicating the number of columns and the format of each column.

The layout of `FunctionCall` and `FunctionCallResponse` messages has changed. `FunctionCall` can now support passing `NULL` arguments to functions. It also can handle passing parameters and retrieving results in either text or binary format. There is no longer any reason to consider `FunctionCall` a potential security hole, since it does not offer direct access to internal server data representations.

The backend sends `ParameterStatus` ('S') messages during connection startup for all parameters it considers interesting to the client library. Subsequently, a `ParameterStatus` message is sent whenever the active value changes for any of these parameters.

The `RowDescription` ('T') message carries new table OID and column number fields for each column of the described row. It also shows the format code for each column.

The `CursorResponse` ('P') message is no longer generated by the backend.

The NotificationResponse ('A') message has an additional string field, which is presently empty but may someday carry additional data passed from the NOTIFY event sender.

The EmptyQueryResponse ('I') message used to include an empty string parameter; this has been removed.

# Capítulo 43. Convenções de codificação do PostgreSQL

## 43.1. Formatação

A formatação do código fonte utiliza um espaçamento de tabulação de 4 colunas, com as tabulações preservadas (ou seja, as tabulações não são expandidas em espaços). Cada nível lógico de recuo (*indentation*) é uma parada adicional de tabulação. As regras de posicionamento (colocação das chaves, etc.) seguem as convenções BSD.

Embora as correções submetidas não sejam obrigadas a seguir, de forma alguma, estas regras de formatação, é uma boa idéia segui-las. O código será processado por `pgindent` e, portanto, não existe motivo para fazê-lo parecer elegante segundo um conjunto de convenções de formatação diferente.

Para o Emacs devem ser adicionadas as seguintes linhas (ou algo semelhante) no arquivo de inicialização `~/ .emacs`:

```
;; verificar arquivos com caminho contendo "postgres" ou "pgsql"
(setq auto-mode-alist
  (cons '("\\(postgres\\|pgsql\\).*\\.\\([ch]\\)" . pgsql-c-mode)
    auto-mode-alist))
(setq auto-mode-alist
  (cons '("\\(postgres\\|pgsql\\).*\\.cc\\)" . pgsql-c-mode)
    auto-mode-alist))

(defun pgsql-c-mode ()
  ;; configurar a formatação para o código C do PostgreSQL
  (interactive)
  (c-mode)
  (setq-default tab-width 4)
  (c-set-style "bsd")           ; definir c-basic-offset como 4, e outras coisas
  (c-set-offset 'case-label '+) ; tweak case indent to match PG custom
  (setq indent-tabs-mode t))    ; manter as tabulações nos recuos
```

Para o editor vi o arquivo `~/ .vimrc`, ou equivalente, deve conter o seguinte:

```
set tabstop=4
```

ou, de maneira equivalente, dentro do vi deve ser utilizado:

```
:set ts=4
```

As ferramentas de exibição de texto `more` e `less` devem ser chamadas utilizando

```
more -x4
less -x4
```

para mostrarem a tabulação de forma apropriada.

## 43.2. Mensagens de erro geradas pelo servidor

As mensagens de erro, de advertência e de `log` geradas a partir do código do servidor devem ser criadas utilizando `ereport`, ou seu primo mais antigo `elog`. A utilização desta função é suficientemente complexa para merecer uma explicação.

Existem dois elementos requeridos em todas as mensagens: o nível de severidade (indo de `DEBUG` até `PANIC`), e o texto primário da mensagem. Além desses, existem elementos opcionais, sendo o mais comum o código de identificação do erro que segue as convenções para `SQLSTATE` da especificação do padrão SQL. O próprio `ereport` é apenas uma casca de função, que existe principalmente pela comodidade sintática de fazer a geração de mensagens se parecer com uma chamada de função no código fonte C. O único parâmetro aceito diretamente pelo `ereport` é o nível de severidade. O texto primário da mensagem, e todos os elementos opcionais da mensagem, são gerados chamando funções auxiliares, como `errmsg`, dentro da chamada a `ereport`.

Uma chamada típica a `ereport` se parece com:

```
ereport(ERROR,
        (errcode(ERRCODE_DIVISION_BY_ZERO),
         errmsg("divisão por zero")));
```

Esta chamada especifica o nível de severidade do erro `ERROR` (um erro comum). A chamada a `errcode` especifica o código de erro `SQLSTATE` utilizando a macro definida em `src/include/utils/errcodes.h`. A chamada a `errmsg` especifica o texto primário da mensagem. Deve-se observar o conjunto extra de parênteses envolvendo as chamadas a funções auxiliares — são aborrecidos mas são sintaticamente necessários.

Abaixo está mostrado um exemplo mais complexo:

```
ereport(ERROR,
        (errcode(ERRCODE_AMBIGUOUS_FUNCTION),
         errmsg("a função %s não é única",
                func_signature_string(funcname, nargs,
                                     actual_arg_types)),
         errhint("Não foi possível escolher a função melhor candidata. "
                 "Pode ser necessário adicionar conversões de tipo explícitas.")));
```

Este exemplo mostra a utilização dos códigos de formato para incorporar valores em tempo de execução ao texto da mensagem; também fornece uma mensagem opcional de “dica” (`hint`).

As rotinas auxiliares disponíveis para `ereport` são:

- `errcode(sqlerrcode)` — especifica o identificador de erro `SQLSTATE` para a condição. Se esta rotina não for chamada, o padrão para o identificador de erro é `ERRCODE_INTERNAL_ERROR` quando o nível de severidade do erro for `ERROR` ou maior, `ERRCODE_WARNING` quando o nível do erro for `WARNING`, senão (para `NOTICE` e abaixo) `ERRCODE_SUCCESSFUL_COMPLETION`. Embora estes padrões sejam convenientes na maioria das vezes, deve-se sempre analisar se são apropriados antes de omitir a chamada a `errcode()`.
- `errmsg(const char *msg, ...)` — especifica o texto primário da mensagem e, possivelmente, valores em tempo de execução a serem inseridos no mesmo. As inserções são especificadas através de códigos de formato no estilo `sprintf`. Além dos códigos de formato padrão aceitos por `sprintf`, pode ser utilizado o código de formato `%m` para inserir a mensagem de erro retornada por `strerror` para o valor corrente de `errno`.<sup>1</sup> `%m` não requer nenhuma entrada associada na lista de parâmetros de `errmsg`. Deve ser observado que a cadeia de caracteres da mensagem é processada por `gettext` para um possível idioma, antes dos códigos de formato serem processados.
- `errmsg_internal(const char *msg, ...)` — é o mesmo que `errmsg`, exceto que a cadeia de caracteres da mensagem não é incluída no dicionário de internacionalização de mensagens. Deve ser utilizada nos casos “que não podem acontecer” e, portanto, provavelmente não vale o esforço necessário para traduzi-la.
- `errdetail(const char *msg, ...)` — produz uma mensagem opcional de “detalhe”; deve ser utilizada quando existe informação adicional, que parece não ser apropriada para ser colocada na mensagem primária. A cadeia de caracteres da mensagem é processada da mesma maneira que em `errmsg`.
- `errhint(const char *msg, ...)` — produz uma mensagem opcional de “dica”; deve ser utilizada para oferecer sugestões sobre como corrigir o problema; o oposto dos detalhes dos fatos sobre o que deu errado. A cadeia de caracteres da mensagem é processada da mesma maneira que em `errmsg`.
- `errcontext(const char *msg, ...)` — normalmente não é chamada diretamente a partir do conjunto de mensagens de `ereport`; em vez disso é utilizada nas funções de chamada (callback) `error_context_stack`, para fornecer informações sobre o contexto onde o erro ocorreu, tal como o local corrente em uma função PL. A cadeia de caracteres da mensagem é processada da mesma maneira que em `errmsg`. Ao contrário das outras funções auxiliares, esta função pode ser chamada mais de uma vez na chamada a `ereport`; as cadeias de caracteres sucessivas fornecidas são concatenadas separadas pelo caractere de nova-linha.
- `errposition(int cursorpos)` — especifica o idioma textual do erro dentro da cadeia de caracteres do comando. Atualmente é útil apenas para os erros detectados nas fases de análise léxica e sintática do processamento do comando.
- `errcode_for_file_access()` — é uma função de conveniência que seleciona o identificador de erro `SQLSTATE` apropriado para uma falha em uma chamada de sistema relacionada com acesso a arquivo. Utiliza o `errno` salvo para determinar o código de erro a ser gerado. Geralmente deve ser utilizada em combinação com `%m` no texto da mensagem de erro primária.

- `errcode_for_socket_access()` — é uma função de conveniência que seleciona o identificador de erro SQLSTATE apropriado para uma falha em uma chamada de sistema relacionada com um soquete.

Existe uma função mais antiga `elog` que ainda é muito utilizada. Uma chamada a `elog`

```
elog(nível, "cadeia de caracteres de formatação", ...);
```

é exatamente equivalente a

```
ereport(nível, (errmsg_internal("cadeia de caracteres de formatação", ...)));
```

Deve ser observado que o código de erro SQLSTATE é sempre o padrão, e que a cadeia de caracteres da mensagem não é incluída no dicionário de internacionalização de mensagens. Portanto, `elog` deve ser utilizada apenas para erros internos e para registro de depuração de baixo nível. Toda mensagem que possivelmente será de interesse dos usuários comuns deve ser emitida através de `ereport`. Apesar disso, existe no sistema um número suficiente de verificação de erros “que não podem acontecer” para que `elog` ainda seja muito utilizada; é preferida para estas mensagens devido à simplicidade de sua notação.

Podem ser encontrados bons conselhos sobre como escrever boas mensagens de erro na Seção 43.3.

### 43.3. Guia de estilo para mensagens de erro

Este guia de estilo é oferecido na esperança de manter um estilo amigável e consistente entre todas as mensagens geradas pelo PostgreSQL.

#### 43.3.1. O que vai aonde

A mensagem primária deve ser curta, baseada em fatos, evitando referências a detalhes da implementação, tal como nomes de funções específicas. “Curta” significa “deve caber em uma linha sob as condições normais”. Se for necessário, deve ser utilizada uma mensagem de detalhe para manter a mensagem primária curta ou mencionar detalhes da implementação, como uma determinada chamada de sistema que falhou. Tanto a mensagem primária quanto a de detalhe devem ser baseadas em fatos. Deve ser utilizada uma mensagem de dica para fazer sugestões sobre o que fazer para corrigir o problema, especialmente se a sugestão não for sempre aplicável.

Por exemplo, em vez de

```
IpMemoryCreate: shmget(chave=%d, tamanho=%u, 0%o) falhou: %m
(mais um longo adendo que é basicamente uma dica)
```

deve ser escrito

```
Primária:    não foi possível criar o segmento de memória compartilhada: %m
Detalhe:     A chamada do sistema que falhou foi shmget(chave=%d, tamanho=%u, 0%o).
Dica:        o adendo
```

Explicação: manter a mensagem primária curta ajuda mantê-la focada, e permite aos clientes organizar o espaço na tela assumindo que uma linha é suficiente para as mensagens de erro. As mensagens de detalhe e de dica podem ser relegadas para o modo verboso ou, talvez, colocadas em uma janela `pop-up` de detalhes de erro. Além disso, os detalhes e as dicas normalmente não devem estar presentes no `log` do servidor para economizar espaço. É melhor evitar as referências aos detalhes de implementação, uma vez que os usuários não o conhecem.

#### 43.3.2. Formatação

No texto das mensagens não se deve supor nada específico com relação à formatação. Deve-se esperar que os clientes e o `log` do servidor quebrem as linhas para ajustá-las às suas próprias necessidades. Nas mensagens longas podem ser utilizados caracteres de nova-linha (`\n`) para indicar quebras de linha sugeridas. A mensagem não deve terminar pelo caractere de nova-linha. Não devem ser utilizados tabulações ou outros caracteres de formatação (quando o contexto do erro é exibido, são adicionados caracteres de nova-linha, automaticamente, para separar os níveis do contexto, tal como chamadas a funções).

Explicação: As mensagens não são mostradas sempre em uma tela de terminal. Nas janelas de Interface Gráfica do Usuário e nos navegadores as instruções de formatação são, na melhor das hipóteses, ignoradas.

### 43.3.3. Aspas

Nos textos em inglês são utilizadas aspas quando é feita uma transcrição. Os textos em outras línguas devem utilizar de forma consistente o sinal indicado pelas normas ortográficas vigentes, e a saída produzida por outros programas de computador.

Explicação: A escolha de aspas em vez de apóstrofes é um tanto arbitrária, mas há uma tendência em se preferir as aspas. Algumas pessoas sugerem escolher o tipo de acento conforme o tipo do objeto de acordo com as convenções da linguagem SQL (ou seja, cadeias de caracteres entre apóstrofes e identificadores entre aspas), mas esta é uma questão técnica interna da linguagem que muitos usuários não estão familiarizados com as mesmas, que não são aplicáveis a outros termos, que não pode ser traduzido para outros idiomas, e que não faz muito sentido também.<sup>2 3</sup>

### 43.3.4. Uso das aspas

Sempre devem ser utilizadas aspas para delimitar nomes de arquivos, identificadores fornecidos pelos usuários e outras variáveis que possam conter palavras. Não devem ser utilizadas aspas em variáveis que não contenham palavras (por exemplo, nomes de operadores).

Existem funções no servidor que duplicam as aspas de suas próprias saídas conforme seja necessário (por exemplo, `format_type_be()`). Não devem ser colocadas aspas adicionais em torno da saída das funções deste tipo.

Explicação: Alguns objetos podem ter nomes que criam ambigüidade quando incorporados ao texto da mensagem. Deve ser indicado de forma consistente onde um nome incorporado começa e termina. A mensagem não deve ficar confusa devido a aspas duplicadas ou desnecessárias.

### 43.3.5. Gramática e pontuação

As regras são diferentes para as mensagens de erro primárias e para as mensagens de detalhe/dica:

Mensagens de erro primárias: a primeira letra não deve ser maiúscula. A mensagem não deve terminar por ponto. De forma alguma a mensagem pode terminar por um ponto de exclamação.

Mensagens de detalhe e de dica: Devem ser utilizados enunciados completos, terminados por ponto. A primeira letra de cada enunciado deve ser maiúscula.

Explicação: Evitar a pontuação torna mais fácil para o aplicativo cliente incorporar a mensagem em vários contextos gramaticais. Geralmente as mensagens primárias não são parágrafos completos (e se forem suficientemente longas para conter mais de um enunciado, devem ser divididas em primária e detalhe). Entretanto, as mensagens de detalhe e de dica são longas e podem precisar incluir vários enunciados. Por consistência devem seguir o estilo de parágrafo completo mesmo quando há apenas um enunciado.

### 43.3.6. Maiúsculas versus Minúsculas

Devem ser utilizadas letras minúsculas nas palavras da mensagem, inclusive a primeira letra da mensagem de erro primária. Devem ser utilizadas letras maiúsculas nos comandos SQL e nas palavras chave que aparecem nas mensagens.

Explicação: É mais fácil fazer que tudo pareça consistente desta forma, uma vez que algumas mensagens são enunciados completos e outras não.

### 43.3.7. Evitar a voz passiva

Deve ser utilizada a voz ativa. Devem ser utilizados enunciados completos quando existir um agente da ação (“A não pode fazer B”). Deve ser utilizado um estilo tipo telegrama sem agente quando o agente é o próprio programa; não utilize “Eu” para o programa.<sup>4 5 6</sup>

Explicação: O programa não é humano. Não finja que seja.

### 43.3.8. Presente versus Passado

Deve ser utilizado o passado quando uma tentativa de fazer algo falhou, mas talvez seja bem sucedida da próxima vez (talvez após a correção de algum problema). Deve ser utilizado o presente se provavelmente a falha é permanente.

Existe uma diferença de semântica que não é trivial entre as formas dos enunciados

não foi possível abrir o arquivo "%s": %m



e

não é possível abrir o arquivo "%s"

A primeira forma significa que uma tentativa de abrir o arquivo falhou. A mensagem deve informar o motivo, tal como “disco cheio” ou “arquivo não existe”. O passado é apropriado porque da próxima vez o disco poderá não estar cheio, ou o arquivo em questão poderá existir.

A segunda forma indica que a funcionalidade de abrir o arquivo especificado não existe no programa, ou que é conceitualmente impossível. O presente é apropriado porque a condição permanecerá indefinidamente.

Explicação: De uma maneira geral o usuário médio não será capaz de chegar a uma conclusão apenas pelo tempo do verbo da mensagem, mas já que a língua portuguesa possui uma gramática esta deve ser utilizada da forma correta.

### 43.3.9. Tipo do objeto

Quando o nome de um objeto é citado, deve ser informado o tipo do objeto.

Explicação: Senão ninguém vai saber a que “foo.bar.baz” se refere.

### 43.3.10. Colchetes

Os colchetes são utilizados apenas em: (1) nas sinopses dos comandos para indicar argumentos opcionais; ou (2) para indicar índice de matriz.

Explicação: Qualquer outra utilização não irá corresponder à utilização comum, só servindo para confundir as pessoas.

### 43.3.11. Montagem das mensagens de erro

Quando uma mensagem inclui texto gerado em outro local, este texto deve ser incorporado usando o estilo:

não foi possível abrir o arquivo %s: %m

Explicação: É difícil levar em consideração todos os códigos de erro possíveis e colocá-los em um único enunciado corrido, portanto será necessário algum tipo de pontuação. Também foi sugerido colocar o texto incorporado entre parênteses, mas isto não é natural quando o texto incorporado pode ser a parte mais importante da mensagem, como geralmente é o caso.

### 43.3.12. Motivos dos erros

As mensagens sempre devem informar o motivo pelo qual o erro ocorreu. Por exemplo:

RUIM: não foi possível abrir o arquivo %s  
MELHOR: não foi possível abrir o arquivo %s (falha de E/S)

Se o motivo for desconhecido, é melhor corrigir o código.

### 43.3.13. Nomes das funções

Não deve ser incluído no texto da mensagem o nome da rotina que está relatando o erro. Existem outros mecanismos para descobrir o nome quando for necessário, e para a maioria dos usuários esta informação não ajuda em nada. Se o texto da mensagem de erro não fizer sentido sem incluir o nome da função, então deve ser reescrito.

RUIM: pg\_atoi: erro em "z": não foi possível analisar "z"  
MELHOR: sintaxe de entrada inválida para inteiro: "z"

Também deve ser evitado mencionar os nomes das funções chamadas; em vez disso, deve ser dito o que o código estava tentando fazer:

RUIM: open() falhou: %m  
MELHOR: não foi possível abrir o arquivo %s: %m

Se realmente for necessário mencionar a chamada de sistema, isto deve ser feito na mensagem de detalhe (Em alguns casos fornecer os verdadeiros valores passados para a chamada de sistema pode ser uma informação apropriada na mensagem de detalhe).

Explicação: Os usuários não sabem o que estas funções fazem.

#### 43.3.14. Palavras ambíguas a serem evitadas

**Incapaz.** “Incapaz” é quase uma passividade. É melhor utilizar “não é possível” ou “não foi possível”, conforme for apropriado.

**Ruim.** As mensagens de erro do tipo “resultado ruim” são realmente difíceis de serem interpretadas de forma inteligente. É melhor escrever porque o resultado foi “ruim” como, por exemplo, “formato inválido”.

**Ilegal.** “Ilegal” significa violação da lei, o resto é “inválido”. Melhor ainda, deve ser dito porque é inválido.

**Desconhecido.** Deve-se tentar evitar o uso de “desconhecido”. Considere o seguinte: “erro: resposta desconhecida”. Se não se sabe qual é a resposta como se sabe que está errada? Geralmente “não reconhecido” é uma escolha melhor. Também deve ser mostrado o valor sobre o qual recai a reclamação.

RUIM: tipo de nó desconhecido

MELHOR: tipo de nó não reconhecido: 42

**Encontrar versus Existir.** Se o programa utilizar um algoritmo não trivial para localizar um recurso (por exemplo, o caminho de procura), e o algoritmo não for bem-sucedido, é justo dizer que o programa não conseguiu “encontrar” o recurso. Se, por outro lado, o local esperado do recurso for conhecido, mas o programa não consegue acessar o recurso neste local, então deve ser dito que o recurso não “existe”. Neste último caso, utilizar “encontrado” soa fraco e confunde o problema.

#### 43.3.15. Escrita apropriada

As palavras devem ser escritas por inteiro. Por exemplo, devem ser evitados:

- espec
- estat
- prog
- aut
- fun

Explicação: Agindo assim melhora a consistência.

#### 43.3.16. Idioma

Deve-se ter em mente que os textos das mensagens de erro precisam ser traduzidos para outros idiomas. Devem ser seguidas as instruções contidas na Seção 44.2.2 para evitar tornar difícil a vida dos tradutores.

## Notas

1. Ou seja, o valor corrente quando a chamada a `ereport` foi encontrada; mudanças em `errno` dentro das rotinas auxiliares não vão afetá-lo. Isto não seria verdade se fosse escrito explicitamente `strerror(errno)` na lista de parâmetros de `errmsg`; por isso, não o faça.
2. O apóstrofo — este sinal ('), que indica supressão de letras, tem hoje o seu emprego bastante reduzido. Usa-se para assinalar: a supressão de uma letra ou mais no verso, por exigência de metrificação; a apócope da vogal *e* em palavras compostas ligadas pela preposição *de* (estrela-d'alva); pronúncias populares ('tá); em derivados de nomes estrangeiros que já têm este sinal. Novo Manual de Português, Celso Pedro Luft, Editora Globo. (N. do T.)
3. As aspas — as aspas ou vírgulas dobradas têm os seguintes empregos: assinalam transcrições textuais; realçam os nomes das obras de arte ou de publicações; caracterizam nomes, intitulativos, apelidos, etc.; marcam expressões, vocábulos, palavras, letras (substantivadas pelo contexto) citadas ou exemplificadas; separam neologismos, estrangeirismos ou quaisquer palavras estranhas ao contexto vernáculo. Novo Manual de Português, Celso Pedro Luft, Editora Globo. (N. do T.)

4. voz ativa — forma em que o verbo se apresenta para normalmente indicar que a pessoa a que se refere é o agente da ação. A pessoa diz-se, neste caso, agente da ação verbal: Eu escrevo a carta, etc. Evanildo Bechara, Moderna Gramática Portuguesa, Edição Revista e Ampliada. (N. do T.)
5. voz passiva — forma verbal que indica que a pessoa é o objeto da ação verbal. A pessoa, neste caso, diz-se paciente da ação verbal: A carta é escrita por mim, etc. Evanildo Bechara, Moderna Gramática Portuguesa, Edição Revista e Ampliada. (N. do T.)
6. voz passiva e passividade — é preciso não confundir voz passiva e passividade. Voz é a forma especial em que se apresenta o verbo para indicar que a pessoa recebe a ação. Passividade é o fato da pessoa receber a ação verbal. Evanildo Bechara, Moderna Gramática Portuguesa, Edição Revista e Ampliada. (N. do T.)

# Capítulo 44. Suporte a idioma nativo

Peter Eisentraut

## 44.1. Para o tradutor

Os programas do PostgreSQL (servidor e cliente) podem emitir as mensagens em seu idioma preferido — se as mensagens tiverem sido traduzidas. Criar e manter um conjunto de mensagens traduzidas requer ajuda de pessoas que falem bem seu próprio idioma, e que desejem contribuir para o trabalho do PostgreSQL. De forma alguma é necessário ser um programador para realizar esta tarefa. Esta seção explica como ajudar.

### 44.1.1. Requisitos

Aqui não será julgado seu conhecimento do idioma — esta seção é sobre ferramentas de software. Teoricamente só é necessário um editor de textos. Mas isto se aplica somente ao caso improvável de não se desejar testar as mensagens traduzidas. Ao configurar a árvore de fontes certifique-se que a opção `--enable-nls` está sendo utilizada. Esta opção também verifica a biblioteca `libintl` e o programa `msgfmt`, que todo usuário final vai necessitar de todo jeito. Para testar a tradução realizada, devem ser seguidas as partes aplicáveis das instruções de instalação.<sup>1</sup>

Se for desejado iniciar um novo trabalho de tradução, ou se for desejado fazer uma mesclagem do catálogo de mensagens (descrita posteriormente), serão necessários os programas `xgettext` e `msgmerge`, respectivamente, em uma implementação compatível com o GNU. Posteriormente isto será arrumado de tal forma que, se for utilizado um pacote de distribuição do código fonte, não será necessário utilizar `xgettext` (A partir do CVS ainda é necessário). É recomendado o GNU Gettext 0.10.36 ou mais recente.

A implementação local do `gettext` vem com a sua própria documentação. Provavelmente parte desta documentação está duplicada no que se segue, mas para obter detalhes adicionais deve ser consultada esta documentação local.

### 44.1.2. Conceitos

Os pares de mensagens original em inglês e equivalente traduzida são mantidos nos *catálogos de mensagens*. Cada programa possui seu próprio catálogo (embora programas relacionados possam compartilhar catálogos de mensagem), sendo um catálogo por idioma. Existem dois formatos para os catálogos de mensagens: O primeiro é o arquivo “PO”, significando “Objeto Portável” (`Portable Object`), que é um arquivo texto puro com sintaxe especial editado pelos tradutores. O segundo é o arquivo “MO”, significando “Objeto de Máquina” (`Machine Object`), que é um arquivo binário gerado a partir do respectivo arquivo PO, usado quando o programa internacionalizado é executado. Os tradutores não lidam com arquivos MO; na verdade quase ninguém lida.

As extensões dos arquivos de catálogos de mensagens são, sem surpresa alguma, `.po` ou `.mo`. O nome base é o nome do programa que o arquivo acompanha, ou o idioma a que se destina, dependendo da situação. Isto confunde um pouco. Por exemplo, `psql.mo` (o arquivo MO do `psql`) ou `pt_BR.po` (arquivo PO em português).

O formato dos arquivos PO está exemplificado abaixo:

```
# comentário

msgid "cadeia de caracteres original"
msgstr "cadeia de caracteres traduzida"

msgid "outra cadeia de caracteres original"
msgstr "outra cadeia de caracteres traduzida"
"as cadeias de caracteres podem ser quebradas desta forma"

...
```

As linhas `msgid` são extraídas do código fonte do programa (Não é necessário, mas esta é a forma mais usada). As linhas `msgstr` são inicialmente vazias, e depois preenchidas com cadeias de caracteres úteis pelo tradutor. As cadeias de caracteres podem conter caracteres de escape no estilo C, e podem ter várias linhas de continuação conforme mostrado acima (A linha de continuação deve começar no início da linha).

O caractere # inicia um comentário. Se o caractere # for seguido imediatamente por um espaço em branco, então este é um comentário mantido pelo tradutor. Podem haver comentários automáticos, que possuem um caractere diferente de espaço em branco logo após o caractere #. Estes comentários são mantidos por várias ferramentas que operam em arquivos PO, e têm por objetivo ajudar o tradutor.

```
#. comentário automático
#: nome_do_arquivo.c:1023
#, sinalizador, sinalizador
```

Os comentários no estilo #. são extraídos do código fonte em que a mensagem é utilizada. Possivelmente o programador inseriu informações para o tradutor, tal como o alinhamento esperado. O comentário #: indica o local exato onde a mensagem é utilizada no código fonte. O tradutor não precisa olhar o código fonte, mas pode olhar se houver dúvidas sobre a tradução correta. Os comentários #, contém sinalizadores que descrevem de alguma forma a mensagem. Atualmente existem dois sinalizadores: *fuzzy* é definido se a mensagem está possivelmente desatualizada devido a alterações no fonte do programa. O tradutor pode verificar e talvez remover o sinalizador *fuzzy*. Deve ser observado que as mensagens sinalizadas como *fuzzy* não se tornam disponíveis para os usuários finais. O outro sinalizador é *c-format*, que indica que a mensagem é um modelo de formato no estilo *printf*. Isto significa que a tradução deve ser capaz de formatar a cadeia de caracteres com o mesmo número e tipo de argumentos. Existem ferramentas que podem fazer a validação.<sup>2 3</sup>

### 44.1.3. Criação e manutenção de catálogos de mensagens

Como fazer para criar um catálogo de mensagens “em branco”? Primeiro, o diretório que contém o programa cujas mensagens se deseja traduzir deve ser tornado o diretório corrente. Se existir o arquivo *nls.mk*, então este programa está preparado para ser traduzido.

Se já existirem alguns arquivos *.po*, então já foi feito algum trabalho de tradução. Os arquivos se chamam *idioma.po*, onde *idioma* é o código de duas letras (minúsculas) especificado em ISO 639-1 (<http://lcweb.loc.gov/standards/iso639-2/englangn.html>) como, por exemplo, *fr.po* para Francês. Se for necessário mais de um trabalho de tradução para o idioma, então os arquivos podem se chamar *idioma\_região.po*, onde *região* é o código de duas letra (maiúsculas) do país especificado em ISO 3166-1 ([http://www.din.de/gremien/nas/nabd/iso3166ma/codlstp1/en\\_listp1.html](http://www.din.de/gremien/nas/nabd/iso3166ma/codlstp1/en_listp1.html)) como, por exemplo, *pt\_BR.po* para o português do Brasil. Se for encontrado o idioma desejado, pode-se simplesmente começar o trabalho a partir deste arquivo.

Se for necessário começar um novo trabalho de tradução, então primeiro deve ser executado o comando

```
gmake init-po
```

para criar o arquivo *prognome.pot*. (*.pot* para diferenciar dos arquivos PO que estão “em produção”. O *T* significa “template” (modelo)). Este arquivo deve ser copiado para *idioma.po* e editado. Para ficar conhecido que existe um novo idioma disponível, deve ser editado o arquivo *nls.mk* e adicionado o código do idioma (ou do idioma e da região), em uma linha parecida com a seguinte:

```
AVAIL_LANGUAGES := cs de es fa fr hu it ko nb pt_BR ro ru sk sl sv tr zh_CN zh_TW
```

(Obviamente, podem estar presentes outros idiomas)

À medida que os programas e bibliotecas subjacentes são modificados, as mensagens podem ser modificadas ou alteradas pelos programadores. Neste caso não é necessário começar do zero. Em vez disso, deve ser executado o comando

```
gmake update-po
```

para criar um novo arquivo de catálogo de mensagens em branco (o arquivo *pot* usado no começo), e mesclá-lo com os arquivos PO existentes. Se o algoritmo de mesclagem não tiver certeza sobre uma determinada mensagem, esta é marcada como “fuzzy” conforme explicado acima. No caso de algo ter dado realmente errado, o arquivo PO antigo é salvo com a extensão *.po.old*.

### 44.1.4. Edição dos arquivos PO

Os arquivos PO podem ser editados usando um editor de textos comum. O tradutor pode apenas mudar o texto entre as aspas após a diretiva *msgstr*, adicionar comentários e alterar o sinalizador *fuzzy*. Existe um modo PO para o Emacs (o que não é surpresa), bastante útil.

Os arquivos PO não precisam ser totalmente preenchidos. O software retorna, automaticamente, a cadeia de caracteres original se não houver nenhuma tradução disponível (ou se a tradução estiver vazia). Não há problema em submeter uma tradução incompleta para ser incluída na árvore de fontes; isto abre espaço para outras pessoas darem continuidade ao trabalho. Entretanto, estimulamos que seja dada prioridade à remoção das entradas `fuzzy` após realizar a mesclagem. Lembre-se que as entradas `fuzzy` não serão instaladas; servem apenas como referência do que seria a tradução correta.

Abaixo seguem recomendações que devem ser lembradas ao editar as traduções:

- Se a mensagem original terminar por nova-linha, certifique-se que a tradução também o faça. Igualmente para tabulações, etc.
- Se a mensagem original for uma cadeia de caracteres no formato `printf`, a tradução também deve ser. A tradução também deve ter os mesmos especificadores de formato e na mesma ordem. Algumas vezes as regras ortográficas tornam isto impossível ou esquisito. Neste caso podem ser mudados os especificadores de formato como mostrado abaixo:

```
msgstr "Die Datei %2$s hat %1$s Zeichen."
```

Desta maneira o primeiro argumento vai, na verdade, utilizar o segundo especificador de formato da lista. A sequência `dígitos$` deve vir imediatamente após sinal de %, antes de qualquer outro manipulador de formato (Esta funcionalidade existe na família de funções `printf`. Não é muito conhecida porque tem pouca utilidade fora da internacionalização de mensagens).

- Se a cadeia de caracteres contiver erro de gramática, o erro deve ser relatado (ou pode ser corrigido no código fonte do programa), e a cadeia de caracteres traduzida normalmente. A cadeia de caracteres corrigida pode ter sido mesclada quando os fontes do programa foram atualizados. Se a cadeia de caracteres original contiver fatos equivocados, isto deve ser relatado (ou pode ser corrigido), não devendo ser traduzida. Em vez disso, a cadeia de caracteres pode ser marcada como comentário no arquivo PO.
- Devem ser mantidos o estilo e o tom da cadeia de caracteres original. Especificamente, as mensagens que não são enunciados (não foi possível abrir o arquivo %s) provavelmente não devem começar por letra maiúscula (se o idioma fizer distinção entre maiúsculas e minúsculas), ou terminar por um ponto (se o idioma utilizar pontuação). Ler a Seção 43.3 pode ajudar.
- Se não for possível descobrir o significado da mensagem, ou se a mensagem for ambígua, pode ser perguntado na lista de discussão dos desenvolvedores o que a mensagem quer dizer. É possível que os usuários finais que falam inglês também não entendam a mensagem ou a considerem ambígua e, assim sendo, é bom melhorar a mensagem.

#### 44.1.5. Exemplos

**Nota:** Seção escrita pelo tradutor, não fazendo parte do manual original.

##### Exemplo 44-1. Personalização das mensagens traduzidas do `psql`

Neste exemplo é mostrado como editar o arquivo `pt_BR.po` que contém as mensagens do `psql` traduzidas por Euler Taveira de Oliveira (<euler@ufgnet.ufg.br>), fazer alguma personalização porventura desejada e, por fim, gerar e colocar o arquivo binário `psql.mo` em produção no Fedora Core 3 e no Windows.

1. Descobrir a versão do `psql` em uso atualmente.  

```
$ psql --version
psql (PostgreSQL) 8.0.1
contém suporte a edição em linha de comando
```
2. Obter em <http://www.postgresql.org/ftp/source/v8.0.1/> o fonte do PostgreSQL correspondente à versão do `psql` (`postgresql-8.0.1.tar.gz`), colocando-o no diretório `/download`, por exemplo.
3. Tornar o diretório `/tmp` o diretório corrente e descompactar o fonte do PostgreSQL.  

```
$ cd /tmp
$ tar xzvf /download/postgresql-8.0.1.tar.gz
```
4. Tornar o diretório de traduções do `psql` o diretório corrente e salvar uma cópia do arquivo traduzido original.  

```
$ cd /tmp/postgresql-8.0.1/src/bin/psql/po/
$ cp pt_BR.po pt_BR.po.original
```
5. Editar o arquivo `pt_BR.po` usando o `gvim`, o `jEdit`, ou outro editor de textos qualquer, mas de preferência o `KBabel`, que é apropriado para traduzir arquivos PO e valida o arquivo traduzido.  

```
$ kbabel pt_BR.po
```

6. Salvar o arquivo `psql.mo` original em produção.

```
$ cp /usr/share/locale/pt_BR/LC_MESSAGES/psql.mo
/usr/share/locale/pt_BR/LC_MESSAGES/psql.mo.original
```

7. Gerar e colocar o arquivo binário `psql.mo` contendo a tradução modificada em produção no Linux.

```
$ msgfmt --statistics -v -c -o psql.mo pt_BR.po
$ mv psql.mo /usr/share/locale/pt_BR/LC_MESSAGES/psql.mo
```

8. Para colocar o arquivo binário `psql.mo` gerado no Linux em produção no Windows, basta copiar este arquivo para o diretório apropriado como, por exemplo, `C:\Arquivos de Programas\PostgreSQL\8.0\share\locale\pt_BR\LC_MESSAGES\psql.mo` ou `E:\Program Files\PostgreSQL\8.0\share\locale\pt_BR\LC_MESSAGES\psql.mo`.

9. Para usar as mensagens traduzidas em uma versão do Windows em inglês, basta configurar o fonte como Lucida Console e definir a página de código e idioma das mensagens no Command Prompt.

```
F:\Documents and Settings\Administrator>E:
E:\>cd \Program Files\PostgreSQL\8.0\bin
E:\Program Files\PostgreSQL\8.0\bin>chcp 1252
Active code page: 1252
E:\Program Files\PostgreSQL\8.0\bin>set LC_MESSAGES=pt_BR
E:\Program Files\PostgreSQL\8.0\bin>psql --version
psql (PostgreSQL) 8.0.1
E:\Program Files\PostgreSQL\8.0\bin>psql -U postgres template1
```

A personalização das mensagens traduzidas do `psql` pode ser toda feita no Windows, utilizando o Winzip ([www.winzip.com](http://www.winzip.com)) para descompactar o arquivo `postgresql-8.0.1.tar.gz`, o poEdit (<http://www.poedit.org/>) para editar o arquivo `pt_BR.po`, e o `msgfmt` para gerar o arquivo `psql.mo`, conforme explicado na nota.

## 44.2. Para o programador

### 44.2.1. Mecânica

Esta seção descreve como implementar suporte a idioma nativo em um programa ou biblioteca que faça parte da distribuição do PostgreSQL. Atualmente somente se aplica a programas C.

#### Adicionar suporte a idioma nativo ao programa

1. O código abaixo deve ser inserido na sequência de inicialização do programa:

```
#ifdef ENABLE_NLS
#include <locale.h>
#endif

...

#ifdef ENABLE_NLS
setlocale(LC_ALL, "");
bindtextdomain("nome_do_programa", LOCALEDIR);
textdomain("nome_do_programa");
#endif
```

(Na verdade o `nome_do_programa` pode ser escolhido livremente)

2. Sempre que for encontrada uma mensagem candidata a ser traduzida deve ser inserida uma chamada a `gettext()`. Por exemplo,

```
fprintf(stderr, "panic level %d\n", lvl);

deve ser alterada para

fprintf(stderr, gettext("panic level %d\n"), lvl);
```

(`gettext` é definida como nenhuma operação se não for configurado suporte a idioma nativo)

Isto tende a tornar o código confuso. Uma forma abreviada comum é:

```
#define _(x) gettext(x)
```

Quando o programa faz grande parte da sua comunicação através de uma ou de poucas funções, tal como `ereport()` no servidor, existe uma outra solução possível. Neste caso faz-se a função chamar `gettext` internamente para todas as cadeias de caracteres.

3. O arquivo `nlsmk` deve ser adicionado ao diretório com os fontes do programa. Este arquivo é lido como um `makefile`. Devem ser feitas as seguintes atribuições de variáveis:

`CATALOG_NAME`

O nome do programa, conforme fornecido na chamada a `textdomain()`.

`AVAIL_LANGUAGES`

Lista das traduções existentes — inicialmente vazia.

`GETTEXT_FILES`

Lista dos arquivos que contêm cadeias de caracteres traduzíveis, ou seja, àquelas marcadas com `gettext` ou por uma solução alternativa. No final esta lista acaba incluindo praticamente todos os arquivos fonte do programa. Se esta lista ficar muito longa, pode ser feito com que o primeiro “arquivo” seja um `+`, e a segunda palavra seja um arquivo que contém um nome de arquivo por linha.

`GETTEXT_TRIGGERS`

As ferramentas que geram catálogos de mensagem para os tradutores trabalharem precisam saber que chamadas de função contêm cadeias de caracteres traduzíveis. Por padrão, só são conhecidas as chamadas a `gettext()`. Se for utilizado `_` ou outros identificadores estes devem ser listados aqui. Se a cadeia de caracteres traduzível não for o primeiro argumento, o item deve estar na forma `func:2` (para o segundo argumento).

O sistema de construção toma conta automaticamente da construção e da instalação dos catálogos de mensagem.

### 44.2.2. Guia para escrever mensagens

Abaixo estão algumas diretrizes para escrever mensagens facilmente traduzíveis.

- Não deve ser construídos enunciados em tempo de execução do tipo:

```
printf("Files were %s.\n", flag ? "copied" : "removed");
```

A ordem das palavras na frase pode ser diferente em outro idioma. Também, mesmo que seja chamado `gettext()` para cada fragmento, os fragmentos separados podem não ficar com uma tradução boa. É melhor duplicar um pequeno código para que cada mensagem traduzida se torne um todo coerente. Somente devem ser inseridos no texto da mensagem em tempo de execução números, nomes de arquivos, e variáveis em tempo de execução deste tipo.

- Por motivos semelhantes, o que vem a seguir também não funciona

```
printf("copied %d file%s", n, n!=1 ? "s" : "");
```

porque assume uma regra de formação de plural. Se houver numeração pode ser resolvido desta maneira

```
if (n==1)
    printf("copied 1 file");
else
    printf("copied %d files", n);
```

e depois ficar desapontado. Alguns idiomas possuem mais de duas formas, com algumas regras peculiares. Pode ser encontrada uma solução para este caso no futuro, mas no momento é melhor evitar este problema. Pode ser escrito:

```
printf("number of copied files: %d", n);
```

- Se for desejado comunicar algo ao tradutor, tal como a mensagem deve se posicionar em relação a outras saídas, a ocorrência da cadeia de caracteres deve ser precedida por um comentário que começa por `translator`, como, por exemplo,

```
/* translator: This message is not what it seems to be. */
```

Estes comentários são copiados para os arquivos de catálogo de mensagens para que os tradutores possam vê-los.



## Notas

1. O msgfmt pode ser utilizado no Windows descompactando gettext-runtime-0.13.1.zip (<http://www.gimp.org/~tml/gimp/win32/gettext-runtime-0.13.1.zip>), gettext-tools-0.13.1.zip (<http://www.gimp.org/~tml/gimp/win32/gettext-tools-0.13.1.zip>) e GNU libiconv (<http://www.gimp.org/~tml/gimp/win32/libiconv-1.9.1.bin.woe32.zip>), presentes na página GTK+ (not GIMP) for Windows (<http://www.gimp.org/~tml/gimp/win32/downloads.html>), por exemplo em E:\gettext, e depois colocando E:\gettext\bin no caminho de procura, ou através do cygwin (<http://www.cygwin.com/>) (N. do T.)
2. O KBabel faz a validação de catálogos PO verificando, inclusive, os argumentos das mensagens com sinalizador `c-format`. (N. do T.)
3. O poEdit (<http://www.poedit.org/>) é um editor de catálogos PO que funciona tanto no Linux quanto no Windows. (N. do T.)

## Capítulo 45. Writing A Procedural Language Handler

All calls to functions that are written in a language other than the current “version 1” interface for compiled languages (this includes functions in user-defined procedural languages, functions written in SQL, and functions using the version 0 compiled language interface), go through a *call handler* function for the specific language. It is the responsibility of the call handler to execute the function in a meaningful way, such as by interpreting the supplied source text. This chapter outlines how a new procedural language's call handler can be written.

The call handler for a procedural language is a “normal” function that must be written in a compiled language such as C, using the version-1 interface, and registered with PostgreSQL as taking no arguments and returning the type `language_handler`. This special pseudotype identifies the function as a call handler and prevents it from being called directly in SQL commands.

The call handler is called in the same way as any other function: It receives a pointer to a `FunctionCallInfoData` struct containing argument values and information about the called function, and it is expected to return a `Datum` result (and possibly set the `isnull` field of the `FunctionCallInfoData` structure, if it wishes to return an SQL null result). The difference between a call handler and an ordinary callee function is that the `flinfo->fn_oid` field of the `FunctionCallInfoData` structure will contain the OID of the actual function to be called, not of the call handler itself. The call handler must use this field to determine which function to execute. Also, the passed argument list has been set up according to the declaration of the target function, not of the call handler.

It's up to the call handler to fetch the entry of the function from the system table `pg_proc` and to analyze the argument and return types of the called function. The `AS` clause from the `CREATE FUNCTION` command for the function will be found in the `prosrc` column of the `pg_proc` row. This is commonly source text in the procedural language, but in theory it could be something else, such as a path name to a file, or anything else that tells the call handler what to do in detail.

Often, the same function is called many times per SQL statement. A call handler can avoid repeated lookups of information about the called function by using the `flinfo->fn_extra` field. This will initially be `NULL`, but can be set by the call handler to point at information about the called function. On subsequent calls, if `flinfo->fn_extra` is already non-`NULL` then it can be used and the information lookup step skipped. The call handler must make sure that `flinfo->fn_extra` is made to point at memory that will live at least until the end of the current query, since an `FmgrInfo` data structure could be kept that long. One way to do this is to allocate the extra data in the memory context specified by `flinfo->fn_mcxt`; such data will normally have the same lifespan as the `FmgrInfo` itself. But the handler could also choose to use a longer-lived memory context so that it can cache function definition information across queries.

When a procedural-language function is invoked as a trigger, no arguments are passed in the usual way, but the `FunctionCallInfoData`'s `context` field points at a `TriggerData` structure, rather than being `NULL` as it is in a plain function call. A language handler should provide mechanisms for procedural-language functions to get at the trigger information.

This is a template for a procedural-language handler written in C:

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
#include "fmgr.h"
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

PG_FUNCTION_INFO_V1(plsample_call_handler);

Datum
plsample_call_handler(PG_FUNCTION_ARGS)
{
    Datum          retval;

    if (CALLED_AS_TRIGGER(fcinfo))
```

```

{
    /*
     * Called as a trigger procedure
     */
    TriggerData    *trigdata = (TriggerData *) fcinfo->context;

    retval = ...
}
else
{
    /*
     * Called as a function
     */

    retval = ...
}

return retval;
}

```

Only a few thousand lines of code have to be added instead of the dots to complete the call handler.

After having compiled the handler function into a loadable module (see Seção 31.9.6), the following commands then register the sample procedural language:

```

CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS 'nome_do_arquivo'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;

```

The procedural languages included in the standard distribution are good references when trying to write your own call handler. Look into the `src/pl` subdirectory of the source tree.

# Capítulo 46. Genetic Query Optimizer

Martin Utesch1997-10-02

**Autor:** Written by Martin Utesch (<utesch@aut.tu-freiberg.de>) for the Institute of Automatic Control at the University of Mining and Technology in Freiberg, Germany.

## 46.1. Query Handling as a Complex Optimization Problem

Among all relational operators the most difficult one to process and optimize is the *join*. The number of alternative plans to answer a query grows exponentially with the number of joins included in it. Further optimization effort is caused by the support of a variety of *join methods* (e.g., nested loop, hash join, merge join in PostgreSQL) to process individual joins and a diversity of *indexes* (e.g., R-tree, B-tree, hash in PostgreSQL) as access paths for relations.

The current PostgreSQL optimizer implementation performs a *near-exhaustive search* over the space of alternative strategies. This algorithm, first introduced in the “System R” database, produces a near-optimal join order, but can take an enormous amount of time and memory space when the number of joins in the query grows large. This makes the ordinary PostgreSQL query optimizer inappropriate for queries that join a large number of tables.

The Institute of Automatic Control at the University of Mining and Technology, in Freiberg, Germany, encountered the described problems as its folks wanted to take the PostgreSQL DBMS as the backend for a decision support knowledge based system for the maintenance of an electrical power grid. The DBMS needed to handle large join queries for the inference machine of the knowledge based system.

Performance difficulties in exploring the space of possible query plans created the demand for a new optimization technique to be developed.

In the following we describe the implementation of a *Genetic Algorithm* to solve the join ordering problem in a manner that is efficient for queries involving large numbers of joins.

## 46.2. Genetic Algorithms

The genetic algorithm (GA) is a heuristic optimization method which operates through nondeterministic, randomized search. The set of possible solutions for the optimization problem is considered as a *population* of *individuals*. The degree of adaptation of an individual to its environment is specified by its *fitness*.

The coordinates of an individual in the search space are represented by *chromosomes*, in essence a set of character strings. A *gene* is a subsection of a chromosome which encodes the value of a single parameter being optimized. Typical encodings for a gene could be *binary* or *integer*.

Through simulation of the evolutionary operations *recombination*, *mutation*, and *selection* new generations of search points are found that show a higher average fitness than their ancestors.

According to the comp.ai.genetic FAQ it cannot be stressed too strongly that a GA is not a pure random search for a solution to a problem. A GA uses stochastic processes, but the result is distinctly non-random (better than random).

### Figura 46-1. Structured Diagram of a Genetic Algorithm

P(t) generation of ancestors at a time t

P''(t) generation of descendants at a time t

[illegible]

### 46.3. Genetic Query Optimization (GEQO) in PostgreSQL

The GEQO module approaches the query optimization problem as though it were the well-known traveling salesman problem (TSP). Possible query plans are encoded as integer strings. Each string represents the join order from one relation of the query to the next. For example, the join tree

$$\begin{array}{c} \diagup \diagdown \\ \diagup \diagdown \quad 2 \\ \diagup \diagdown \quad 3 \\ 4 \quad 1 \end{array}$$

is encoded by the integer string '4-1-3-2', which means, first join relation '4' and '1', then '3', and then '2', where 1, 2, 3, 4 are relation IDs within the PostgreSQL optimizer.

Parts of the GEQO module are adapted from D. Whitley's Genitor algorithm.

Specific characteristics of the GEQO implementation in PostgreSQL are:

- Usage of a *steady state* GA (replacement of the least fit individuals in a population, not whole-generational replacement) allows fast convergence towards improved query plans. This is essential for query handling with reasonable time;
- Usage of *edge recombination crossover* which is especially suited to keep edge losses low for the solution of the TSP by means of a GA;
- Mutation as genetic operator is deprecated so that no repair mechanisms are needed to generate legal TSP tours.

The GEQO module allows the PostgreSQL query optimizer to support large join queries effectively through non-exhaustive search.

### 46.3.1. Future Implementation Tasks for PostgreSQL GEQO

Work is still needed to improve the genetic algorithm parameter settings. In file `src/backend/optimizer/geqo/geqo_main.c`, routines `gimme_pool_size` and `gimme_number_generations`, we have to find a compromise for the parameter settings to satisfy two competing demands:

- Optimality of the query plan
- Computing time

At a more basic level, it is not clear that solving query optimization with a GA algorithm designed for TSP is appropriate. In the TSP case, the cost associated with any substring (partial tour) is independent of the rest of the tour, but this is

certainly not true for query optimization. Thus it is questionable whether edge recombination crossover is the most effective mutation procedure.

## 46.4. Further Reading

The following resources contain additional information about genetic algorithms:

- The Hitch-Hiker's Guide to Evolutionary Computation (<http://surf.de.uu.net/encore/www/>) (FAQ for comp.ai.genetic (news://comp.ai.genetic))
- Evolutionary Computation and its application to art and design (<http://www.red3d.com/cwr/evolve.html>) by Craig Reynolds
- *Sistemas de Banco de Dados*
- *The design and implementation of the POSTGRES query optimizer*

# Capítulo 47. Index Cost Estimation Functions

**Autor:** Written by Tom Lane (<tgl@sss.pgh.pa.us>) on 2000-01-24

**Nota:** This must eventually become part of a much larger chapter about writing new index access methods.

Every index access method must provide a cost estimation function for use by the planner/optimizer. The procedure OID of this function is given in the `amcostestimate` field of the access method's `pg_am` entry.

**Nota:** Prior to PostgreSQL 7.0, a different scheme was used for registering index-specific cost estimation functions.

The `amcostestimate` function is given a list of WHERE clauses that have been determined to be usable with the index. It must return estimates of the cost of accessing the index and the selectivity of the WHERE clauses (that is, the fraction of main-table rows that will be retrieved during the index scan). For simple cases, nearly all the work of the cost estimator can be done by calling standard routines in the optimizer; the point of having an `amcostestimate` function is to allow index access methods to provide index-type-specific knowledge, in case it is possible to improve on the standard estimates.

Each `amcostestimate` function must have the signature:

```
void
amcostestimate (Query *root,
                RelOptInfo *rel,
                IndexOptInfo *index,
                List *indexQuals,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation);
```

The first four parameters are inputs:

`root`

The query being processed.

`rel`

The relation the index is on.

`index`

The index itself.

`indexQuals`

List of index qual clauses (implicitly ANDed); a NIL list indicates no qualifiers are available.

The last four parameters are pass-by-reference outputs:

`*indexStartupCost`

Set to cost of index start-up processing

`*indexTotalCost`

Set to total cost of index processing

`*indexSelectivity`

Set to index selectivity

`*indexCorrelation`

Set to correlation coefficient between index scan order and underlying table's order

Note that cost estimate functions must be written in C, not in SQL or any available procedural language, because they must access internal data structures of the planner/optimizer.

The index access costs should be computed in the units used by `src/backend/optimizer/path/costsize.c`: a sequential disk block fetch has cost 1.0, a nonsequential fetch has cost `random_page_cost`, and the cost of processing one index row should usually be taken as `cpu_index_tuple_cost` (which is a user-adjustable optimizer parameter). In addition, an appropriate multiple of `cpu_operator_cost` should be charged for any comparison operators invoked during index processing (especially evaluation of the indexQuals themselves).

The access costs should include all disk and CPU costs associated with scanning the index itself, but NOT the costs of retrieving or processing the main-table rows that are identified by the index.

The “start-up cost” is the part of the total scan cost that must be expended before we can begin to fetch the first row. For most indexes this can be taken as zero, but an index type with a high start-up cost might want to set it nonzero.

The `indexSelectivity` should be set to the estimated fraction of the main table rows that will be retrieved during the index scan. In the case of a lossy index, this will typically be higher than the fraction of rows that actually pass the given qual conditions.

The `indexCorrelation` should be set to the correlation (ranging between -1.0 and 1.0) between the index order and the table order. This is used to adjust the estimate for the cost of fetching rows from the main table.

## Cost Estimation

A typical cost estimator will proceed as follows:

1. Estimate and return the fraction of main-table rows that will be visited based on the given qual conditions. In the absence of any index-type-specific knowledge, use the standard optimizer function `clauselist_selectivity()`:

```
*indexSelectivity = clauselist_selectivity(root, indexQuals,
                                         rel->relid, JOIN_INNER);
```

2. Estimate the number of index rows that will be visited during the scan. For many index types this is the same as `indexSelectivity` times the number of rows in the index, but it might be more. (Note that the index's size in pages and rows is available from the `IndexOptInfo` struct.)
3. Estimate the number of index pages that will be retrieved during the scan. This might be just `indexSelectivity` times the index's size in pages.
4. Compute the index access cost. A generic estimator might do this:

```
/*
 * Our generic assumption is that the index pages will be read
 * sequentially, so they have cost 1.0 each, not random_page_cost.
 * Also, we charge for evaluation of the indexquals at each index row.
 * All the costs are assumed to be paid incrementally during the scan.
 */
cost_qual_eval(&index_qual_cost, indexQuals);
*indexStartupCost = index_qual_cost.startup;
*indexTotalCost = numIndexPages +
    (cpu_index_tuple_cost + index_qual_cost.per_tuple) * numIndexTuples;
```

5. Estimate the index correlation. For a simple ordered index on a single field, this can be retrieved from `pg_statistic`. If the correlation is not known, the conservative estimate is zero (no correlation).

Examples of cost estimator functions can be found in `src/backend/utils/adts/selfuncs.c`.

By convention, the `pg_proc` entry for an `amcostestimate` function should show eight arguments all declared as `internal` (since none of them have types that are known to SQL), and the return type is `void`.



# Capítulo 48. GiST Indexes

## 48.1. Introdução

GiST stands for Generalized Search Tree. It is a balanced, tree-structured access method, that acts as a base template in which to implement arbitrary indexing schemes. B+-trees, R-trees and many other indexing schemes can be implemented in GiST.

One advantage of GiST is that it allows the development of custom data types with the appropriate access methods, by an expert in the domain of the data type, rather than a database expert.

Some of the information here is derived from the University of California at Berkeley's GiST Indexing Project web site (<http://gist.cs.berkeley.edu/>) and Marcel Kornacker's thesis, Access Methods for Next-Generation Database Systems (<http://citeseer.nj.nec.com/448594.html>). The GiST implementation in PostgreSQL is primarily maintained by Teodor Sigaev and Oleg Bartunov, and there is more information on their website: <http://www.sai.msu.su/~megera/postgres/gist/>.

## 48.2. Extensibility

Traditionally, implementing a new index access method meant a lot of difficult work. It was necessary to understand the inner workings of the database, such as the lock manager and Write-Ahead Log. The GiST interface has a high level of abstraction, requiring the access method implementor to only implement the semantics of the data type being accessed. The GiST layer itself takes care of concurrency, logging and searching the tree structure.

This extensibility should not be confused with the extensibility of the other standard search trees in terms of the data they can handle. For example, PostgreSQL supports extensible B+-trees and R-trees. That means that you can use PostgreSQL to build a B+-tree or R-tree over any data type you want. But B+-trees only support range predicates ( $<$ ,  $=$ ,  $>$ ), and R-trees only support n-D range queries (contains, contained, equals).

So if you index, say, an image collection with a PostgreSQL B+-tree, you can only issue queries such as “is image equal to image”, “is image less than image” and “is image greater than image”? Depending on how you define “equals”, “less than” and “greater than” in this context, this could be useful. However, by using a GiST based index, you could create ways to ask domain-specific questions, perhaps “find all images of horses” or “find all over-exposed images”.

All it takes to get a GiST access method up and running is to implement seven user-defined methods, which define the behavior of keys in the tree. Of course these methods have to be pretty fancy to support fancy queries, but for all the standard queries (B+-trees, R-trees, etc.) they're relatively straightforward. In short, GiST combines extensibility along with generality, code reuse, and a clean interface.

## 48.3. Implementation

There are seven methods that an index operator class for GiST must provide:

consistent

Given a predicate  $p$  on a tree page, and a user query,  $q$ , this method will return false if it is certain that both  $p$  and  $q$  cannot be true for a given data item.

union

This method consolidates information in the tree. Given a set of entries, this function generates a new predicate that is true for all the entries.

compress

Converts the data item into a format suitable for physical storage in an index page.

decompress

The reverse of the `compress` method. Converts the index representation of the data item into a format that can be manipulated by the database.

`penalty`

Returns a value indicating the “cost” of inserting the new entry into a particular branch of the tree. items will be inserted down the path of least `penalty` in the tree.

`picksplit`

When a page split is necessary, this function decides which entries on the page are to stay on the old page, and which are to move to the new page.

`same`

Returns true if two entries are identical, false otherwise.

## 48.4. Limitations

The current implementation of GiST within PostgreSQL has some major limitations: GiST access is not concurrent; the GiST interface doesn't allow the development of certain data types, such as digital trees (see papers by Aoki et al); and there is not yet any support for write-ahead logging of updates in GiST indexes.

Solutions to the concurrency problems appear in Marcel Kornacker's thesis; however these ideas have not yet been put into practice in the PostgreSQL implementation.

The lack of write-ahead logging is just a small matter of programming, but since it isn't done yet, a crash could render a GiST index inconsistent, forcing a REINDEX.

## 48.5. Exemplos

To see example implementations of index methods implemented using GiST, examine the following contrib modules:

`btree_gist`

B-Tree

`cube`

Indexing for multi-dimensional cubes

`intarray`

RD-Tree for one-dimensional array of int4 values

`ltree`

Indexing for tree-like stuctures

`rtree_gist`

R-Tree

`seg`

Storage and indexed access for “float ranges”

`tsearch` and `tsearch2`

Full text indexing

# Capítulo 49. Armazenamento físico dos bancos de dados

Este capítulo fornece uma visão geral do formato de armazenamento físico utilizado nos bancos de dados do PostgreSQL.

## 49.1. Organização dos arquivos de banco de dados

Esta seção descreve o formato de armazenamento no nível de arquivos e diretórios.

Todos os dados necessários para um agrupamento de bancos de dados são armazenados dentro do diretório de dados do agrupamento, geralmente referenciado como `PGDATA` (devido ao nome da variável de ambiente que pode ser utilizada para defini-lo). Um local comum para `PGDATA` é `/var/lib/pgsql/data`. Podem existir na mesma máquina vários agrupamentos, gerenciados por diferentes `postmaster`.

O diretório `PGDATA` contém vários subdiretórios e arquivos de controle, conforme mostrado na Tabela 49-1. Além destes itens requeridos, os arquivos de configuração do agrupamento `postgresql.conf`, `pg_hba.conf` e `pg_ident.conf` são tradicionalmente armazenados em `PGDATA` (embora a partir da versão 8.0 do PostgreSQL seja possível mantê-los em qualquer outro lugar).

**Tabela 49-1. Conteúdo de PGDATA**

Item	Descrição
<code>PG_VERSION</code>	Arquivo contendo o número de versão principal do PostgreSQL
<code>base</code>	Subdiretório contendo subdiretórios por banco de dados
<code>global</code>	Subdiretório contendo tabelas para todo o agrupamento, como <code>pg_database</code>
<code>pg_clog</code>	Subdiretório contendo dados sobre status de efetivação de transação
<code>pg_subtrans</code>	Subdiretório contendo dados sobre status de subtransação
<code>pg_tblspc</code>	Subdiretório contendo vínculos simbólicos para espaços de tabelas
<code>pg_xlog</code>	Subdiretório contendo os arquivos do WAL (registro prévio da escrita)
<code>postmaster.opts</code>	Arquivo contendo as opções de linha de comando com as quais o <code>postmaster</code> foi inicializado da última vez
<code>postmaster.pid</code>	Arquivo de bloqueio contendo o PID corrente do <code>postmaster</code> , e o ID do segmento de memória compartilhada (não mais presente após o <code>postmaster</code> ser parado)

Para cada banco de dados do agrupamento existe um subdiretório dentro de `PGDATA/base`, com nome correspondente ao OID do banco de dados em `pg_database`. Este subdiretório é o local padrão para os arquivos do banco de dados; em particular, os catálogos do sistema do banco de dados são armazenados neste subdiretório.

Cada tabela e índice é armazenado em um arquivo separado, com nome correspondente ao número do *filenode* da tabela ou do índice, que pode ser encontrado em `pg_class.relfilenode`.

### Cuidado

Deve ser observado que enquanto o *filenode* da tabela geralmente corresponde ao seu OID, *não* é necessariamente assim; algumas operações, como `TRUNCATE`, `REINDEX`, `CLUSTER`, e algumas formas de `ALTER TABLE`, podem mudar o *filenode* e preservar o OID. Deve-se evitar assumir que o *filenode* e o OID da tabela sejam idênticos.

Quando uma tabela ou um índice excede 1Gb, este é dividido em *segmentos* de até 1 GB. O nome do primeiro arquivo de segmento é o mesmo do *filenode*; os arquivos subseqüentes são chamados de *filenode.1*, *filenode.2*, etc. Esta organização evita problemas em plataformas que possuem limitação de tamanho de arquivo. O conteúdo das tabelas e dos índices são discutidos em mais detalhes na Seção 49.3.

As tabelas que possuem colunas com entradas potencialmente grandes possuem uma tabela *TOAST* (fatias) associada, que é utilizada para armazenamento fora-de-linha dos valores de campo que são muito grandes para serem mantidos na própria linha da tabela. `pg_class.reltoastrelid` faz o vínculo entre a tabela e a sua tabela *TOAST*, caso haja alguma. Para obter informações adicionais deve ser consultada a Seção 49.2.

Os espaços de tabela tornam o cenário mais complicado. Cada espaço de tabelas definido pelo usuário possui um vínculo simbólico dentro do diretório `PGDATA/pg_tblspc`, que aponta para o diretório físico do espaço de tabelas (conforme especificado em seu comando `CREATE TABLESPACE`). O nome do vínculo simbólico corresponde ao OID do espaço de tabelas. Dentro do diretório físico do espaço de tabelas existe um subdiretório, para cada banco de dados que possui elementos no espaço de tabelas, com nome correspondente ao OID do banco de dados. As tabelas dentro deste diretório seguem o esquema de nomes baseado no *filenode*. O espaço de tabelas `pg_default` não é acessado através de `pg_tblspc`, e corresponde a `PGDATA/base`. De maneira semelhante, o espaço de tabelas `pg_global` não é acessado através de `pg_tblspc`, e corresponde a `PGDATA/global`.

## 49.2. TOAST

Esta seção fornece uma visão geral do *TOAST* (A técnica de armazenamento de atributo de tamanho grande).

Uma vez que o PostgreSQL utiliza um tamanho de página fixo (geralmente 8Kb), e não permite as tuplas se estenderem por várias páginas, não é possível armazenar valores de campo muito grandes diretamente. Antes da versão 7.1 do PostgreSQL, havia um limite rígido de apenas uma página para a quantidade total de dados que podia ser colocado em uma linha da tabela. Na versão 7.1, e posteriores, este limite é superado permitindo os valores dos campos serem comprimidos e/ou divididos em várias linhas físicas. Isto acontece de forma transparente para o usuário, causando apenas um pequeno impacto para a maior parte do código do servidor. Esta técnica é afetuosamente chamada de *TOAST* (ou “a melhor coisa desde o pão em fatias”).

Apenas certos tipos de dado suportam *TOAST* — não é necessário impor sobrecarga em tipos de dado que não podem produzir valores de campo grandes. Para suportar o *TOAST*, o tipo de dado deve possuir uma representação de comprimento variável (*varlena*), na qual a primeira palavra de 32 bits de qualquer valor armazenado contém o comprimento total do valor em bytes (incluindo a si próprio). O *TOAST* não restringe o restante da representação. Todas as funções no nível-C com suporte a tipo de dado fatiável devem tomar o cuidado de tratar os valores de entrada na forma de *TOAST* (Geralmente isto é feito chamando `PG_DETOAST_DATUM` antes de fazer qualquer coisa com o valor de entrada; mas em alguns casos é possível uma abordagem mais eficiente).

*TOAST* se apodera dos dois bits de mais alta ordem da palavra de comprimento *varlena*, limitando, portanto, o tamanho lógico de qualquer valor de um tipo de dado fatiável a 1Gb ( $2^{30}$  - 1 bytes). Quando os dois bits são iguais a zero, o valor é um valor comum não fatiado do tipo de dado. Um desses bits, se estiver definido, indica que o valor foi comprimido devendo ser descomprimido antes de ser utilizado. O outro bit, se estiver definido, indica que o valor foi armazenado fora-de-linha. Neste caso, o restante do valor é apenas um ponteiro e os dados corretos devem ser encontrados em outro lugar. Quando os dois bits estão definidos, os dados fora-de-linha também estão comprimidos. Em cada caso o comprimento nos bits de mais baixa ordem da palavra *varlena* indica o tamanho real do dado, e não o tamanho do valor lógico que seria obtido pela descompressão ou busca dos dados fora-de-linha.

Quando alguma coluna da tabela é fatiável, a tabela possui uma tabela *TOAST* associada, cujo OID é armazenado na entrada `pg_class.reltoastrelid` da tabela. Os valores fatiados fora-de-linha são mantidos na tabela *TOAST*, conforme descrito em mais detalhes abaixo.

A técnica de compressão utilizada é um membro bem simples e bem rápido da família de técnicas de compressão *LZ*. Para obter detalhes deve ser visto o arquivo `src/backend/utils/adt/pg_lzcompress.c`.

Os valores fora-de-linha são divididos (após a compressão, se esta for aplicada) em pedaços de no máximo `TOAST_MAX_CHUNK_SIZE` bytes (este valor é um pouco menor que `BLCKSZ/4`, ou cerca de 2000 bytes por padrão). Cada pedaço é armazenado como uma linha separada na tabela *TOAST* para a tabela possuidora. Toda tabela *TOAST* possui as colunas `chunk_id` (OID identificador de um determinado valor fatiado), `chunk_seq` (número de sequência do pedaço dentro de seu valor) e `chunk_data` (os dados reais do pedaço). Um índice único englobando `chunk_id` e `chunk_seq` permite a busca rápida dos valores. O dado ponteiro que representa o valor fora-de-linha fatiado necessita, portanto, armazenar o OID da tabela *TOAST* onde é feita a procura, e o OID do valor específico (seu `chunk_id`). Por conveniência, o dado ponteiro também armazena o tamanho do dado lógico (comprimento original do dado não comprimido), e o tamanho real armazenado (diferente, caso tenha sido aplicada compressão). Incluindo a palavra de cabeçalho *varlena*, o tamanho total do dado ponteiro para *TOAST* é portanto de 20 bytes, a despeito do tamanho real do valor representado.

O código TOAST é disparado apenas quando um valor de linha a ser armazenado na tabela é maior que `BLCKSZ/4` bytes (normalmente 2Kb). O código TOAST comprime e/ou move os valores de campo para fora-de-linha até que o valor da linha se torne menor `BLCKSZ/4` bytes, ou que não possa mais obter ganho. Durante a operação de atualização, os valores não alterados dos campos normalmente são preservados na forma em que estão; portanto, a atualização de uma linha com valores fora-de-linha não incorre em custos de TOAST, se nenhum dos valores fora-de-linha for alterado.

O código TOAST reconhece quatro estratégias diferentes para armazenar colunas fatiáveis:

- `PLAIN` não permite compressão ou armazenamento fora-de-linha. Esta é a única estratégia possível para as colunas com tipo de dado não fatiável.
- `EXTENDED` permite tanto compressão quanto armazenamento fora-de-linha. Este é o padrão para a maioria dos tipos de dado fatiáveis. Primeiro tenta-se a compressão, e depois o armazenamento fora-de-linha se a linha continuar muito grande.
- `EXTERNAL` permite armazenamento fora-de-linha, mas não a compressão. A utilização de `EXTERNAL` faz com que as operações em subcadeias de caracteres nas colunas com tipos de dado `text` e `bytea` sejam mais rápidas (ao custo de um maior espaço de armazenamento), porque estas operações são otimizadas para buscar apenas as partes requeridas do valor fora-de-linha quando este não está comprimido.
- `MAIN` permite compressão, mas não permite o armazenamento fora-de-linha (Na verdade o armazenamento fora-de-linha ainda será feito para estas colunas, mas apenas como último recurso quando não houver maneira de tornar a linha pequena o suficiente).

Cada tipo de dado fatiável especifica a estratégia padrão para as colunas deste tipo de dado, mas a estratégia para uma determinada coluna de uma tabela pode ser alterada pelo comando `ALTER TABLE SET STORAGE`.

Este esquema possui várias vantagens quando comparado com uma abordagem mais direta, como a que permite os valores de linha se estenderem por várias páginas. Assumindo que os comandos geralmente são qualificados por comparações com valores chave relativamente pequenos, a maior parte do trabalho do executor é feito utilizando a entrada principal da linha. Os valores grandes dos atributos fatiados serão trazidos (se forem selecionados), somente na hora em que o conjunto contendo os resultados for enviado para o cliente. Portanto, a tabela principal é muito menor, cabendo uma quantidade maior de suas linhas no cache de buffers compartilhados do que caberia no caso de não haver o armazenamento fora-de-linha. Os conjuntos de classificação também encolhem, e a classificação será feita inteiramente em memória com mais frequência. Um pequeno teste mostrou que uma tabela contendo páginas HTML típicas, e suas respectivas URLs, foi armazenada na metade do seu tamanho bruto incluindo a tabela TOAST, e que a tabela principal continha apenas 10% de todos os dados (As URLs e algumas páginas HTML pequenas). Não houve diferença no tempo de execução quando comparada com a tabela não fatiada, onde todas as páginas HTML foram cortadas para caber em 7Kb.

### 49.3. Disposição das páginas de banco de dados

Esta seção fornece uma visão geral do formato de página utilizado dentro das tabelas e índices do PostgreSQL.<sup>1</sup> As tabelas de sequência e TOAST são formatadas como qualquer outra tabela regular.

Na explicação a seguir, é assumido que um *byte* tem 8 bits. Além disso, o termo *item* se refere a um valor de dado individual armazenado na página. Em uma tabela, um item é uma linha; em um índice, um item é uma entrada do índice.

Todas as tabelas e índices são armazenadas em uma matriz de *páginas* de tamanho fixo (geralmente 8Kb, embora possa ser selecionado um tamanho de página diferente ao compilar o servidor). Em uma tabela todas as páginas são logicamente equivalentes, portanto um determinado item (linha) pode ser armazenado em qualquer página. Nos índices, a primeira página geralmente é reservada para uma *metapágina* contendo informações de controle, e podem existir tipos diferentes de página dentro do índice, dependendo do método de acesso do índice.

A Tabela 49-2 mostra a disposição global da página. Existem cinco partes em cada página.

**Tabela 49-2. Disposição global da página**

Item	Descrição
PageHeaderData	Comprimento de 20 bytes. Contém informações gerais sobre a página, incluindo ponteiros para espaços livres.
ItemPointerData	Matriz de pares (deslocamento, comprimento) apontando para os itens existentes. 4 bytes por item.

Item	Descrição
Espaço livre	Espaço não alocado. Os novos ponteiros de item são alocados a partir do início desta área, e os novos itens a partir do fim.
Itens	Os próprios itens existentes.
Espaço especial	Dados específicos do método de acesso do índice. Métodos diferentes armazenam dados diferentes. Vazio nas tabelas comuns.

Os primeiros 20 bytes de cada página compõem o cabeçalho da página (PageHeaderData). Seu formato é detalhado na Tabela 49-3. Os primeiros dois campos registram a entrada mais recente no WAL relacionada a esta página. São seguidos por três campos inteiros de 2 bytes (`pd_lower`, `pd_upper`, e `pd_special`). Estes campos contêm o deslocamento em bytes do início da página ao início do espaço não alocado, ao final do espaço não alocado, e ao início do espaço especial. Os últimos 2 bytes do cabeçalho da página, `pd_pagesize_version`, armazenam o tamanho da página e o indicador de versão. A partir do PostgreSQL 8.0 o número da versão é 2; O PostgreSQL 7.3 e 7.4 usam a versão número 1; as versões anteriores usam a versão número 0 (A disposição básica da página e o formato do cabeçalho não mudaram nestas versões, mas a disposição dos cabeçalhos das linhas `heap` mudou). Basicamente, o tamanho da página somente está presente como uma verificação cruzada; não há suporte para a existência de mais de um tamanho de página em uma instalação.

Tabela 49-3. Disposição de PageHeaderData

Campo	Tipo	Comprimento	Descrição
<code>pd_lsn</code>	XLogRecPtr	8 bytes	LSN: próximo byte após o último byte do registro do xlog (gerenciador do registro de transação do PostgreSQL) para a última modificação nesta página
<code>pd_tli</code>	TimeLineID	4 bytes	TLI da última mudança
<code>pd_lower</code>	LocationIndex	2 bytes	Deslocamento até o início do espaço livre
<code>pd_upper</code>	LocationIndex	2 bytes	Deslocamento até o final do espaço livre
<code>pd_special</code>	LocationIndex	2 bytes	Deslocamento até o início do espaço especial
<code>pd_pagesize_version</code>	uint16	2 bytes	Informação sobre o tamanho em bytes e número da versão de disposição da página

Todos os detalhes podem ser encontrados no arquivo `src/include/storage/bufpage.h`.<sup>2</sup>

```
/*
 * A página de disco do postgres é uma camada de abstração por cima do
 * bloco de disco do postgres (que é simplesmente uma unidade de E/S)
 * (veja block.h).
 *
 * Especificamente, enquanto um bloco de disco pode estar não-formatado,
 * uma página de disco é sempre uma página com encaixes na forma:
 *
 * +-----+-----+-----+-----+
 * | PageHeaderData | linp1 linp2 linp3 ... |
 * +-----+-----+-----+-----+
 * | ... linpN | |
 * +-----+-----+-----+-----+
 * |           ^ pd_lower |
 * |           |           |
 * |           v pd_upper |
 * +-----+-----+-----+-----+
 * |           | tuplaN ... |
 * +-----+-----+-----+-----+
 * |           ... tupla3 tupla2 tupla1 | "espaço especial" |
 * +-----+-----+-----+-----+
```

```

*                                     ^ pd_special
*
* a página fica cheia quando não pode ser adicionado mais nada entre
* pd_lower e pd_upper.
*
* todos os blocos escritos por um método de acesso devem ser páginas de disco.
*
* EXCEÇÕES:
*
* como é óbvio, a página não é formatada antes de ser inicializada por uma
* chamada a PageInit.
*
* NOTAS:
*
* linp1..N formam uma matriz de ItemId. Os ItemPointers apontam para esta
* matriz em vez de apontar diretamente para a tupla. Deve ser observado que
* OffsetNumbers convencionalmente começa por 1, e não por 0.
*
* tupla1..N são adicionadas na página "de trás para frente".
* Como o ItemPointer da tupla aponta para a sua entrada ItemId, em vez de
* apontar para a sua posição real medida em deslocamento em bytes, as tuplas
* podem ser fisicamente embaralhadas na página sempre que houver necessidade.
*
* Informações genéricas por página do método de acesso são mantidas em
* PageHeaderData.
*
* Informações específicas por página do método de acesso (se existirem) são
* mantidas na área marcada como "espaço especial"; cada método de acesso
* possui uma estrutura "opaca" definida em algum lugar que é armazenada como
* o rodapé da página. O método de acesso deve inicializar sempre suas páginas
* com PageInit, e depois definir seus próprios campos opacos.
*/

```

```
typedef Pointer Page;
```

```

/*
* posição (deslocamento em bytes) dentro da página.
*
* deve ser observado que na verdade está limitado a 2^15, porque
* ItemIdData.lp_off e ItemIdData.lp_len foram limitados a 15 bits
* (veja itemid.h).
*/

```

```
typedef uint16 LocationIndex;
```

```

/*
* organização da página do disco
*
* informações de gerenciamento de espaço genéricas para qualquer página
*
* pd_lsn
*     identifica o registro do xlog para a última modificação nesta página.
* pd_tli
*     a mesma coisa.
* pd_lower
*     deslocamento até o início do espaço livre
* pd_upper
*     deslocamento até o final do espaço livre
* pd_special
*     deslocamento até o início do espaço especial
* pd_pagesize_version

```

```

*      tamanho em bytes e número da versão de disposição da página
*
* O LSN é utilizado pelo gerenciador de buffers para garantir a regra
* básica do WAL: "deve ser escrito em xlog antes de escrever os dados".
* Um buffer sujo não pode ser descarregado no disco até que xlog tenha
* sido descarregado atingindo pelo menos o LSN da página.
* O TLI também é armazenado para fins de identificação (não é claro se é
* realmente necessário, mas parece ser uma boa idéia).
*
* O número da versão da página e o tamanho da página são empacotados juntos
* em um único campo uint16. Isto se deve a motivos históricos: antes do
* PostgreSQL 7.3, não havia o conceito de número de versão de página,
* e fazendo desta maneira fingimos que os bancos de dados anteriores a
* versão 7.3 possuem o número de versão de página zero.
* Os tamanhos de página são restritos a múltiplos de 256, deixando os 8 bits
* de mais baixa ordem disponíveis para o número da versão.
*
* O tamanho mínimo possível de uma página é, talvez, 64B para caber o
* cabeçalho da página, o espaço opaco e uma tupla mínima; obviamente,
* na realidade se deseja um tamanho muito maior e, portanto, a restrição
* do tamanho da página ser múltiplo de 256 não é uma restrição importante.
* Do lado mais alto, só é possível suportar páginas de até 32KB, porque
* lp_off/lp_len são 15 bits.
*/
typedef struct PageHeaderData
{
    /* XXX LSN é membro de *qualquer* bloco, não apenas os organizados por página */
    XLogRecPtr      pd_lsn;          /* LSN: próximo byte após o último byte
                                     * do registro do xlog para a última
                                     * modificação nesta página */

    TimeLineID      pd_tli;          /* TLI da última modificação */
    LocationIndex    pd_lower;        /* Deslocamento até o início de espaço livre */
    LocationIndex    pd_upper;        /* Deslocamento até o final de espaço livre */
    LocationIndex    pd_special;      /* Deslocamento até o início de espaço especial */
    uint16           pd_pagesize_version;
    ItemIdData       pd_linp[1];     /* Início da matriz de ponteiro de linha */
} PageHeaderData;

typedef PageHeaderData *PageHeader;

/*
* O número de versão de disposição de página 0 é usado nas versões
* anteriores a 7.3 do Postgres. As versões 7.3 e 7.4 utilizam 1,
* denotando a nova disposição de HeapTupleHeader. A versão 8.0 mudou
* a disposição de HeapTupleHeader novamente.
*/
#define PG_PAGE_LAYOUT_VERSION      2

```

Depois do cabeçalho da página estão os identificadores de itens (`ItemIdData`), cada um requerendo quatro bytes. O identificador de item contém o deslocamento em bytes até o início do item, o comprimento do item em bytes, e uns poucos bits de atributo que afetam a interpretação do item. Os novos identificadores são alocados, conforme a necessidade, a partir do início do espaço não alocado. O número de itens identificadores presentes pode ser determinado olhando `pd_lower`, que é aumentado para abrir espaço para o novo identificador. Como um identificador de item nunca é movido até que seja liberado, seu índice pode ser utilizado por longo prazo para referenciar um item, mesmo quando o próprio item é movido dentro da página para compactar o espaço livre. Na verdade, todo ponteiro para um item (`ItemPointer`, também conhecido como `CTID`) criado pelo PostgreSQL, consiste de um número de página e o índice do identificador do item.

Os itens em si são armazenados em um espaço obtido de trás para frente a partir do final do espaço não alocado. A estrutura exata varia conforme o conteúdo da tabela. As tabelas e as seqüências utilizam uma estrutura chamada `HeapTupleHeaderData`, descrita abaixo.



A seção final é a “seção especial”, e pode conter qualquer coisa que o método de acesso deseje armazenar. Por exemplo, os índices *b-tree* armazenam vínculos para os irmãos (*siblings*) esquerdo e direito da página, assim como alguns outros dados relevantes para a estrutura do índice. As tabelas comuns não utilizam a seção especial (indicado pela definição `pd_special` igual ao tamanho da página).

Todas as linhas da tabela são estruturadas da mesma maneira. Existe um cabeçalho de tamanho fixo (ocupando 27 bytes na maioria das máquinas), seguido por um mapa de bits de nulo opcional, um campo do ID do objeto opcional, e os dados do usuário. O cabeçalho está detalhado na Tabela 49-4. Os dados verdadeiros (colunas da linha) começam no primeiro deslocamento indicado por `t_hoff`, que sempre deve ser um múltiplo da distância `MAXALIGN` para a plataforma. O mapa de bits de nulo somente está presente se o bit `HEAP_HASNULL` estiver definido em `t_infomask`. Se estiver presente, começa logo após o cabeçalho fixo e ocupa uma quantidade de bytes suficiente para ter um bit para cada coluna de dados (ou seja, `t_natts` bits no total). Nesta lista de bits, o bit 1 indica não-nulo, e o bit 0 indica nulo. Quando o mapa de bits não está presente, é assumido que o valor de todas as colunas é diferente de nulo (não-nulas). O ID do objeto só está presente quando o bit `HEAP_HASOID` está definido em `t_infomask`. Se estiver presente, aparece logo antes da fronteira de `t_hoff`. Qualquer enchimento necessário para tornar `t_hoff` um múltiplo de `MAXALIGN` fica entre o mapa de bits de nulo e o ID do objeto (Por sua vez, isto garante que o ID do objeto está alinhado de forma apropriada).

**Tabela 49-4. Disposição de HeapTupleHeaderData**

Campo	Tipo	Comprimento	Descrição
<code>t_xmin</code>	TransactionId	4 bytes	marca do XID de inserção
<code>t_cmin</code>	CommandId	4 bytes	marca do CID de inserção
<code>t_xmax</code>	TransactionId	4 bytes	marca do XID de exclusão
<code>t_cmax</code>	CommandId	4 bytes	marca do CID de exclusão (sobrepõe <code>t_xvac</code> )
<code>t_xvac</code>	TransactionId	4 bytes	XID da operação de VACUUM movendo a versão da linha
<code>t_ctid</code>	ItemPointerData	6 bytes	TID corrente desta ou de uma nova versão da linha
<code>t_natts</code>	int16	2 bytes	número de atributos
<code>t_infomask</code>	uint16	2 bytes	vários bits sinalizadores
<code>t_hoff</code>	uint8	1 byte	deslocamento até os dados do usuário

Todos os detalhes se encontram em `src/include/access/htup.h`.

A interpretação dos dados reais somente pode ser feita com informações obtidas a partir de outras tabelas, principalmente `pg_attribute`. Os valores chave necessários para identificar a posição do campo são `attlen` e `attalign`. Não há maneira de obter um determinado atributo diretamente, exceto quando existem somente campos de largura fixa e nenhum nulo. Todos os truques estão contidos nas funções `heap_getattr`, `fastgetattr` e `heap_getsysattr`.

Para ler os dados é necessário examinar cada atributo por vez. Primeiro deve ser verificado se o campo é nulo utilizando o mapa de bits de nulo. Se for, deve-se ir para o próximo. Depois deve haver certeza de estar no alinhamento correto. Se o campo for de largura fixa, então todos os bytes estão simplesmente colocados em seus lugares. Se for um campo de largura variável (`attlen = -1`), então é um pouco mais complicado. Todos os tipos de dado de comprimento variável compartilham uma estrutura de cabeçalho comum, `varattrib`, que inclui o comprimento total do valor armazenado e alguns bits sinalizadores. Dependendo dos sinalizadores, os dados podem estar em-linha ou em uma tabela TOAST; também pode estar comprimido (consulte a Seção 49.2).

## Notas

1. Na verdade, os métodos de acesso de índice não precisam utilizar este formato de página. Todos os métodos de índice existentes utilizam este formato básico, mas os dados mantidos nas metapáginas dos índices geralmente não seguem as regras de disposição de item.
2. O trecho do arquivo `bufpage.h` mostrado abaixo não faz parte do manual original. (N. do T.)

# Capítulo 50. BKI Backend Interface

Backend Interface (BKI) files are scripts in a special language that is understood by the PostgreSQL backend when running in the “bootstrap” mode. The bootstrap mode allows system catalogs to be created and filled from scratch, whereas ordinary SQL commands require the catalogs to exist already. BKI files can therefore be used to create the database system in the first place. (And they are probably not useful for anything else.)

initdb uses a BKI file to do part of its job when creating a new database cluster. The input file used by initdb is created as part of building and installing PostgreSQL by a program named `genbki.sh`, which reads some specially formatted C header files in the `src/include/catalog/` directory of the source tree. The created BKI file is called `postgres.bki` and is normally installed in the `share` subdirectory of the installation tree.

Related information may be found in the documentation for initdb.

## 50.1. BKI File Format

This section describes how the PostgreSQL backend interprets BKI files. This description will be easier to understand if the `postgres.bki` file is at hand as an example.

BKI input consists of a sequence of commands. Commands are made up of a number of tokens, depending on the syntax of the command. Tokens are usually separated by whitespace, but need not be if there is no ambiguity. There is no special command separator; the next token that syntactically cannot belong to the preceding command starts a new one. (Usually you would put a new command on a new line, for clarity.) Tokens can be certain key words, special characters (parentheses, commas, etc.), numbers, or double-quoted strings. Everything is case sensitive.

Lines starting with a `#` are ignored.

## 50.2. BKI Commands

`create [bootstrap] [shared_relation] [without_oids] nome_da_tabela (nome1 = tipo1 [, nome2 = type2, ...])`

Create a table named *tablename* with the columns given in parentheses.

The following column types are supported directly by `bootstrap.c`: `bool`, `bytea`, `char` (1 byte), `name`, `int2`, `int4`, `regproc`, `regclass`, `regtype`, `text`, `oid`, `tid`, `xid`, `cid`, `int2vector`, `oidvector`, `_int4` (array), `_text` (array), `_aclitem` (array). Although it is possible to create tables containing columns of other types, this cannot be done until after `pg_type` has been created and filled with appropriate entries.

When `bootstrap` is specified, the table will only be created on disk; nothing is entered into `pg_class`, `pg_attribute`, etc, for it. Thus the table will not be accessible by ordinary SQL operations until such entries are made the hard way (with `insert` commands). This option is used for creating `pg_class` etc themselves.

The table is created as shared if `shared_relation` is specified. It will have OIDs unless `without_oids` is specified.

`open nome_da_tabela`

Open the table called *nome\_da\_tabela* for further manipulation.

`close [nome_da_tabela]`

Close the open table called *tablename*. It is an error if *nome\_da\_tabela* is not already opened. If no *tablename* is given, then the currently open table is closed.

`insert [OID = valor_do_oid] (valor1 valor2 ...)`

Insert a new row into the open table using *value1*, *value2*, etc., for its column values and *oid\_value* for its OID. If *valor\_do\_oid* is zero (0) or the clause is omitted, then the next available OID is used.

NULL values can be specified using the special key word `_null_`. Values containing spaces must be double quoted.

declare [unique] index *indexname* on *tablename* using *amname* (*opclass1 name1* [, ...])

Create an index named *indexname* on the table named *nome\_da\_tabela* using the *amname* access method. The fields to index are called *name1*, *name2* etc., and the operator classes to use are *opclass1*, *opclass2* etc., respectively. The index file is created and appropriate catalog entries are made for it, but the index contents are not initialized by this command.

build indices

Fill in the indices that have previously been declared.

### 50.3. Example

The following sequence of commands will create the table `test_table` with two columns `cola` and `colb` of type `int4` and `text`, respectively, and insert two rows into the table.

```
create test_table (cola = int4, colb = text)
open test_table
insert OID=421 ( 1 "value1" )
insert OID=422 ( 2 _null_ )
close test_table
```

## **VIII. Apêndices**

# Apêndice A. Códigos de erro do PostgreSQL

Toda mensagem emitida pelo servidor PostgreSQL possui um código de erro de cinco caracteres associado, que segue as convenções do padrão SQL para códigos de “SQLSTATE”. Os aplicativos que precisam saber qual condição de erro ocorreu devem, normalmente, testar o código de erro em vez de analisar o texto da mensagem de erro. Os códigos de erro têm menos probabilidade de mudar entre versões do PostgreSQL e, também, não estão sujeitos a mudanças devido ao idioma das mensagens de erro (pt\_BR, etc.). Deve ser observado que alguns códigos de erro gerados pelo PostgreSQL estão definidos no padrão SQL, mas não todos; alguns códigos de erro adicionais, para condições não definidas no padrão, foram criados ou tomados emprestado de outros gerenciadores de banco de dados.

De acordo com o padrão, os dois primeiros caracteres do código de erro denotam a classe do erro, enquanto os três últimos caracteres indicam uma condição específica dentro da classe. Portanto, um aplicativo que não reconhece um determinado código de erro ainda pode ser capaz de inferir o que fazer a partir da classe do erro.

A Tabela A-1 lista todos os códigos de erro definidos no PostgreSQL 8.0.0 (Na verdade alguns não são utilizados atualmente, mas são definidos pelo padrão SQL). As classes de erro também estão mostradas. Para cada classe de erro existe um código de erro “padrão”, possuindo os três últimos caracteres iguais a 000. Este código é utilizado apenas para as condições de erro que se enquadram na classe, mas que não possuem nenhum código mais específico atribuído.

No PL/pgSQL o nome da condição associada ao código de erro é idêntico à frase mostrada na tabela, com espaços substituídos pelo caractere sublinhado. Por exemplo, o nome da condição associada ao código 22012, `DIVISION BY ZERO`, é `DIVISION_BY_ZERO`. Os nomes das condições podem ser escritos tanto com letras maiúscula quanto com letras minúsculas (Deve ser observado que o PL/pgSQL não reconhece os nomes das condições de advertência, que são as classes 00, 01 e 02, ao contrário dos erros).

**Tabela A-1. Códigos de erro do PostgreSQL**

Código de erro	Significado
Class 00	Successful Completion
00000	SUCCESSFUL COMPLETION
Class 01	Warning
01000	WARNING
0100C	DYNAMIC RESULT SETS RETURNED
01008	IMPLICIT ZERO BIT PADDING
01003	NULL VALUE ELIMINATED IN SET FUNCTION
01007	PRIVILEGE NOT GRANTED
01006	PRIVILEGE NOT REVOKED
01004	STRING DATA RIGHT TRUNCATION
01P01	DEPRECATED FEATURE
Class 02	No Data — this is also a warning class per SQL:1999
02000	NO DATA
02001	NO ADDITIONAL DYNAMIC RESULT SETS RETURNED
Class 03	SQL Statement Not Yet Complete
03000	SQL STATEMENT NOT YET COMPLETE
Class 08	Connection Exception
08000	CONNECTION EXCEPTION

Código de erro	Significado
08003	CONNECTION DOES NOT EXIST
08006	CONNECTION FAILURE
08001	SQLCLIENT UNABLE TO ESTABLISH SQLCONNECTION
08004	SQLSERVER REJECTED ESTABLISHMENT OF SQLCONNECTION
08007	TRANSACTION RESOLUTION UNKNOWN
08P01	PROTOCOL VIOLATION
Class 09	Triggered Action Exception
09000	TRIGGERED ACTION EXCEPTION
Class 0A	Feature Not Supported
0A000	FEATURE NOT SUPPORTED
Class 0B	Invalid Transaction Initiation
0B000	INVALID TRANSACTION INITIATION
Class 0F	Locator Exception
0F000	LOCATOR EXCEPTION
0F001	INVALID LOCATOR SPECIFICATION
Class 0L	Invalid Grantor
0L000	INVALID GRANTOR
0LP01	INVALID GRANT OPERATION
Class 0P	Invalid Role Specification
0P000	INVALID ROLE SPECIFICATION
Class 21	Cardinality Violation
21000	CARDINALITY VIOLATION
Class 22	Data Exception
22000	DATA EXCEPTION
2202E	ARRAY SUBSCRIPT ERROR
22021	CHARACTER NOT IN REPERTOIRE
22008	DATETIME FIELD OVERFLOW
22012	DIVISION BY ZERO
22005	ERROR IN ASSIGNMENT
2200B	ESCAPE CHARACTER CONFLICT
22022	INDICATOR OVERFLOW
22015	INTERVAL FIELD OVERFLOW
2201E	INVALID ARGUMENT FOR LOGARITHM
2201F	INVALID ARGUMENT FOR POWER FUNCTION
2201G	INVALID ARGUMENT FOR WIDTH BUCKET FUNCTION

Código de erro	Significado
22018	INVALID CHARACTER VALUE FOR CAST
22007	INVALID DATETIME FORMAT
22019	INVALID ESCAPE CHARACTER
2200D	INVALID ESCAPE OCTET
22025	INVALID ESCAPE SEQUENCE
22010	INVALID INDICATOR PARAMETER VALUE
22020	INVALID LIMIT VALUE
22023	INVALID PARAMETER VALUE
2201B	INVALID REGULAR EXPRESSION
22009	INVALID TIME ZONE DISPLACEMENT VALUE
2200C	INVALID USE OF ESCAPE CHARACTER
2200G	MOST SPECIFIC TYPE MISMATCH
22004	NULL VALUE NOT ALLOWED
22002	NULL VALUE NO INDICATOR PARAMETER
22003	NUMERIC VALUE OUT OF RANGE
22026	STRING DATA LENGTH MISMATCH
22001	STRING DATA RIGHT TRUNCATION
22011	SUBSTRING ERROR
22027	TRIM ERROR
22024	UNTERMINATED C STRING
2200F	ZERO LENGTH CHARACTER STRING
22P01	FLOATING POINT EXCEPTION
22P02	INVALID TEXT REPRESENTATION
22P03	INVALID BINARY REPRESENTATION
22P04	BAD COPY FILE FORMAT
22P05	UNTRANSLATABLE CHARACTER
Class 23	Integrity Constraint Violation
23000	INTEGRITY CONSTRAINT VIOLATION
23001	RESTRICT VIOLATION
23502	NOT NULL VIOLATION
23503	FOREIGN KEY VIOLATION
23505	UNIQUE VIOLATION
23514	CHECK VIOLATION
Class 24	Invalid Cursor State
24000	INVALID CURSOR STATE

Código de erro	Significado
Class 25	Invalid Transaction State
25000	INVALID TRANSACTION STATE
25001	ACTIVE SQL TRANSACTION
25002	BRANCH TRANSACTION ALREADY ACTIVE
25008	HELD CURSOR REQUIRES SAME ISOLATION LEVEL
25003	INAPPROPRIATE ACCESS MODE FOR BRANCH TRANSACTION
25004	INAPPROPRIATE ISOLATION LEVEL FOR BRANCH TRANSACTION
25005	NO ACTIVE SQL TRANSACTION FOR BRANCH TRANSACTION
25006	READ ONLY SQL TRANSACTION
25007	SCHEMA AND DATA STATEMENT MIXING NOT SUPPORTED
25P01	NO ACTIVE SQL TRANSACTION
25P02	IN FAILED SQL TRANSACTION
Class 26	Invalid SQL Statement Name
26000	INVALID SQL STATEMENT NAME
Class 27	Triggered Data Change Violation
27000	TRIGGERED DATA CHANGE VIOLATION
Class 28	Invalid Authorization Specification
28000	INVALID AUTHORIZATION SPECIFICATION
Class 2B	Dependent Privilege Descriptors Still Exist
2B000	DEPENDENT PRIVILEGE DESCRIPTORS STILL EXIST
2BP01	DEPENDENT OBJECTS STILL EXIST
Class 2D	Invalid Transaction Termination
2D000	INVALID TRANSACTION TERMINATION
Class 2F	SQL Routine Exception
2F000	SQL ROUTINE EXCEPTION
2F005	FUNCTION EXECUTED NO RETURN STATEMENT
2F002	MODIFYING SQL DATA NOT PERMITTED
2F003	PROHIBITED SQL STATEMENT ATTEMPTED
2F004	READING SQL DATA NOT PERMITTED
Class 34	Invalid Cursor Name
34000	INVALID CURSOR NAME
Class 38	External Routine Exception
38000	EXTERNAL ROUTINE EXCEPTION
38001	CONTAINING SQL NOT PERMITTED
38002	MODIFYING SQL DATA NOT PERMITTED



Código de erro	Significado
38003	PROHIBITED SQL STATEMENT ATTEMPTED
38004	READING SQL DATA NOT PERMITTED
Class 39	External Routine Invocation Exception
39000	EXTERNAL ROUTINE INVOCATION EXCEPTION
39001	INVALID SQLSTATE RETURNED
39004	NULL VALUE NOT ALLOWED
39P01	TRIGGER PROTOCOL VIOLATED
39P02	SRF PROTOCOL VIOLATED
Class 3B	Savepoint Exception
3B000	SAVEPOINT EXCEPTION
3B001	INVALID SAVEPOINT SPECIFICATION
Class 3D	Invalid Catalog Name
3D000	INVALID CATALOG NAME
Class 3F	Invalid Schema Name
3F000	INVALID SCHEMA NAME
Class 40	Transaction Rollback
40000	TRANSACTION ROLLBACK
40002	TRANSACTION INTEGRITY CONSTRAINT VIOLATION
40001	SERIALIZATION FAILURE
40003	STATEMENT COMPLETION UNKNOWN
40P01	DEADLOCK DETECTED
Class 42	Syntax Error or Access Rule Violation
42000	SYNTAX ERROR OR ACCESS RULE VIOLATION
42601	SYNTAX ERROR
42501	INSUFFICIENT PRIVILEGE
42846	CANNOT COERCE
42803	GROUPING ERROR
42830	INVALID FOREIGN KEY
42602	INVALID NAME
42622	NAME TOO LONG
42939	RESERVED NAME
42804	DATATYPE MISMATCH
42P18	INDETERMINATE DATATYPE
42809	WRONG OBJECT TYPE
42703	UNDEFINED COLUMN

Código de erro	Significado
42883	UNDEFINED FUNCTION
42P01	UNDEFINED TABLE
42P02	UNDEFINED PARAMETER
42704	UNDEFINED OBJECT
42701	DUPLICATE COLUMN
42P03	DUPLICATE CURSOR
42P04	DUPLICATE DATABASE
42723	DUPLICATE FUNCTION
42P05	DUPLICATE PREPARED STATEMENT
42P06	DUPLICATE SCHEMA
42P07	DUPLICATE TABLE
42712	DUPLICATE ALIAS
42710	DUPLICATE OBJECT
42702	AMBIGUOUS COLUMN
42725	AMBIGUOUS FUNCTION
42P08	AMBIGUOUS PARAMETER
42P09	AMBIGUOUS ALIAS
42P10	INVALID COLUMN REFERENCE
42611	INVALID COLUMN DEFINITION
42P11	INVALID CURSOR DEFINITION
42P12	INVALID DATABASE DEFINITION
42P13	INVALID FUNCTION DEFINITION
42P14	INVALID PREPARED STATEMENT DEFINITION
42P15	INVALID SCHEMA DEFINITION
42P16	INVALID TABLE DEFINITION
42P17	INVALID OBJECT DEFINITION
Class 44	WITH CHECK OPTION Violation
44000	WITH CHECK OPTION VIOLATION
Class 53	Insufficient Resources
53000	INSUFFICIENT RESOURCES
53100	DISK FULL
53200	OUT OF MEMORY
53300	TOO MANY CONNECTIONS
Class 54	Program Limit Exceeded
54000	PROGRAM LIMIT EXCEEDED

Código de erro	Significado
54001	STATEMENT TOO COMPLEX
54011	TOO MANY COLUMNS
54023	TOO MANY ARGUMENTS
Class 55	Object Not In Prerequisite State
55000	OBJECT NOT IN PREREQUISITE STATE
55006	OBJECT IN USE
55P02	CANT CHANGE RUNTIME PARAM
55P03	LOCK NOT AVAILABLE
Class 57	Operator Intervention
57000	OPERATOR INTERVENTION
57014	QUERY CANCELED
57P01	ADMIN SHUTDOWN
57P02	CRASH SHUTDOWN
57P03	CANNOT CONNECT NOW
Class 58	System Error (errors external to PostgreSQL itself)
58030	IO ERROR
58P01	UNDEFINED FILE
58P02	DUPLICATE FILE
Class F0	Configuration File Error
F0000	CONFIG FILE ERROR
F0001	LOCK FILE EXISTS
Class P0	PL/pgSQL Error
P0000	PLPGSQL ERROR
P0001	RAISE EXCEPTION
Class XX	Internal Error
XX000	INTERNAL ERROR
XX001	DATA CORRUPTED
XX002	INDEX CORRUPTED

## Apêndice B. Apoio a data e hora

O PostgreSQL utiliza um analisador heurístico interno para dar suporte a toda entrada de data e hora. Data e hora são entradas como cadeias de caracteres, e divididas em campos distintos com base na determinação preliminar do tipo de informação que pode estar contida no campo. Cada campo é interpretado e, em seguida, atribuído um valor numérico, ignorado, ou rejeitado. O analisador contém tabelas internas de procura para todos os campos textuais, incluindo meses, dias da semana e zonas horárias.

Este apêndice inclui informações sobre o conteúdo destas tabelas de procura, e descreve os passos utilizados pelo analisador para decodificar data e hora.

### B.1. Interpretação de data e hora

As entradas dos tipos data e hora são todas decodificadas utilizando o seguinte procedimento.

1. Dividir a cadeia de caracteres da entrada em elementos (*tokens*), e categorizar cada elemento como sendo uma cadeia de caracteres, hora, zona horária ou número.
  - a. Se o elemento numérico contiver dois-pontos (:), então é uma cadeia de caracteres de hora. Incluir todos os dígitos e dois-pontos subsequentes.
  - b. Se o elemento numérico contiver hífen (-), barra (/), ou dois ou mais pontos (.), então é uma cadeia de caracteres de data que pode conter o mês na forma de texto.
  - c. Se o elemento contiver apenas números, então é um campo único ou uma data ISO 8601 concatenada (por exemplo, 19990113 para 13 de janeiro de 1999), ou hora (por exemplo, 141516 para 14:15:16).
  - d. Se o elemento começar por um sinal de mais (+), ou de menos (-), então é uma zona horária ou um campo especial.
2. Se o elemento for uma cadeia de caracteres, estabelecer correspondência com as cadeias de caracteres possíveis.
  - a. Fazer uma pesquisa binária do elemento na tabela, para ver se é uma cadeia de caracteres especial (por exemplo, *today*), um dia da semana (por exemplo, *Thursday*), um mês (por exemplo, *January*), ou uma palavra sem efeito (por exemplo, *at*, *on*).

Definir valor e máscara de bit para os campos. Por exemplo, definir ano, mês e dia para *today* e, adicionalmente, hora, minuto e segundo para *now*.
  - b. Se não for encontrado, fazer uma pesquisa binária semelhante na tabela para fazer a correspondência entre o elemento e uma zona horária.
  - c. Se ainda não for encontrado, lançar um erro.
3. Quando o elemento é um número ou campo numérico:
  - a. Se contiver 8 ou 6 dígitos, e nenhum outro campo de data foi lido anteriormente, então interpretar como uma “data concatenada” (por exemplo, 19990118 ou 990118). A interpretação é *YYYYMMDD* ou *YYMMDD*.
  - b. Se o elemento tiver 3 dígitos, e um ano já tiver sido lido, então interpretar como dia do ano.
  - c. Se contiver 4 ou 6 dígitos, e o ano já tiver sido lido, então interpretar como hora (*HHMM* ou *HHMMSS*).
  - d. Se contiver 3 ou mais dígitos, e nenhum campo data foi encontrado anteriormente, interpretar como ano (isto força a ordem *yy-mm-dd* para os demais campos de data).
  - e. Senão, a ordem do campo data é assumida como definida por *DateStyle*: *mm-dd-yy*, *dd-mm-yy*, ou *yy-mm-dd*. Lançar um erro se o campo mês ou dia estiver fora do intervalo permitido.
4. Se for especificado BC, tornar o ano negativo e adicionar um para armazenamento interno (Não existe ano zero no Calendário Gregoriano <sup>1</sup> e, portanto, numericamente 1 BC se torna o ano zero).
5. Se não for especificado BC, e o campo do ano tiver comprimento de dois dígitos, então ajustar o ano para quatro dígitos. Se o campo for inferior a 70, então adicionar 2000, senão adicionar 1900.

**Dica:** Os anos Gregorianos AD 1-99 <sup>2</sup> podem ser entrados utilizando 4 dígitos com zeros à esquerda (por exemplo, 0099 é AD 99). As versões anteriores do PostgreSQL aceitavam anos com três dígitos e com um dígito, mas a partir da versão 7.0 as regras ficaram mais rigorosas, para reduzir a possibilidade de ambigüidade.

## B.2. Palavras chave para data e hora

A Tabela B-1 mostra os elementos reconhecidos como nomes dos meses.

**Tabela B-1. Nomes dos meses**

Nome	Abreviatura
January	Jan
February	Feb
March	Mar
April	Apr
May	
June	Jun
July	Jul
August	Aug
September	Sep, Sept
October	Oct
November	Nov
December	Dec

A Tabela B-2 mostra os elementos reconhecidos como nomes dos dias da semana.

**Tabela B-2. Nomes dos dias da semana**

Nome	Abreviatura
Sunday	Sun
Monday	Mon
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri
Saturday	Sat

A Tabela B-3 mostra os elementos que servem como modificadores para várias finalidades.

**Tabela B-3. Modificadores de campo de data e hora**

Identificador	Descrição
ABSTIME	Ignorado
AM	Hora antes das 12:00

Identificador	Descrição
AT	Ignorado
JULIAN, JD, J	O próximo campo é dia Juliano
ON	Ignorado
PM	Hora após ou às 12:00
T	O próximo campo é hora

A palavra chave `ABSTIME` é ignorada por razões históricas; em versões muito antigas do PostgreSQL, valores inválidos do tipo `abstime` eram informados como `Invalid Abstime`. Entretanto, este não é mais o caso, e esta palavra chave provavelmente será retirada em uma versão futura.

A Tabela B-4 mostra as abreviaturas de zona horária reconhecidas pelo PostgreSQL nos valores de entrada de data e hora. Deve ser observado que estes nomes *não* são necessariamente usados na saída de data e hora — a saída é ditada pela abreviatura oficial de zona horária associada no momento à definição do parâmetro `timezone` (É provável que as versões futuras façam algum uso de `timezone` para a entrada também).

A tabela está organizada pelo deslocamento da zona horária com relação à UTC, em vez de alfabeticamente, com a finalidade de facilitar a correspondência entre a utilização local e as abreviaturas reconhecidas nos casos em que forem diferentes.

**Tabela B-4. Abreviaturas das zonas horárias para a entrada**

Zona horária	Deslocamento da UTC	Descrição
NZDT	+13:00	New Zealand Daylight-Saving Time
IDLE	+12:00	International Date Line, East
NZST	+12:00	New Zealand Standard Time
NZT	+12:00	New Zealand Time
AESST	+11:00	Australia Eastern Summer Standard Time
ACSST	+10:30	Central Australia Summer Standard Time
CADT	+10:30	Central Australia Daylight-Saving Time
SADT	+10:30	South Australian Daylight-Saving Time
AEST	+10:00	Australia Eastern Standard Time
EAST	+10:00	East Australian Standard Time
GST	+10:00	Guam Standard Time, Russia zone 9
LIGT	+10:00	Melbourne, Australia
SAST	+09:30	South Australia Standard Time
CAST	+09:30	Central Australia Standard Time
AWSST	+09:00	Australia Western Summer Standard Time
JST	+09:00	Japan Standard Time, Russia zone 8
KST	+09:00	Korea Standard Time
MHT	+09:00	Kwajalein Time
WDT	+09:00	West Australian Daylight-Saving Time
MT	+08:30	Moluccas Time

Zona horária	Deslocamento da UTC	Descrição
AWST	+08:00	Australia Western Standard Time
CCT	+08:00	China Coastal Time
WADT	+08:00	West Australian Daylight-Saving Time
WST	+08:00	West Australian Standard Time
JT	+07:30	Java Time
ALMST	+07:00	Almaty Summer Time
WAST	+07:00	West Australian Standard Time
CXT	+07:00	Christmas (Island) Time
MMT	+06:30	Myanmar Time
ALMT	+06:00	Almaty Time
MAWT	+06:00	Mawson (Antarctica) Time
IOT	+05:00	Indian Chagos Time
MVT	+05:00	Maldives Island Time
TFT	+05:00	Kerguelen Time
AFT	+04:30	Afghanistan Time
EAST	+04:00	Antananarivo Summer Time
MUT	+04:00	Mauritius Island Time
RET	+04:00	Reunion Island Time
SCT	+04:00	Mahe Island Time
IRT, IT	+03:30	Iran Time
EAT	+03:00	Antananarivo, Comoro Time
BT	+03:00	Baghdad Time
EETDST	+03:00	Eastern Europe Daylight-Saving Time
HMT	+03:00	Hellas Mediterranean Time (?)
BDST	+02:00	British Double Summer Time
CEST	+02:00	Central European Summer Time
CETDST	+02:00	Central European Daylight-Saving Time
EET	+02:00	Eastern European Time, Russia zone 1
FWT	+02:00	French Winter Time
IST	+02:00	Israel Standard Time
MEST	+02:00	Middle European Summer Time
METDST	+02:00	Middle Europe Daylight-Saving Time
SST	+02:00	Swedish Summer Time
BST	+01:00	British Summer Time
CET	+01:00	Central European Time

Zona horária	Deslocamento da UTC	Descrição
DNT	+01:00	<i>Dansk Normal Tid</i>
FST	+01:00	French Summer Time
MET	+01:00	Middle European Time
MEWT	+01:00	Middle European Winter Time
MEZ	+01:00	<i>Mitteleuropäische Zeit</i>
NOR	+01:00	Norway Standard Time
SET	+01:00	Seychelles Time
SWT	+01:00	Swedish Winter Time
WETDST	+01:00	Western European Daylight-Saving Time
GMT	00:00	Greenwich Mean Time
UT	00:00	Universal Time
UTC	00:00	Universal Coordinated Time
Z	00:00	Same as UTC
ZULU	00:00	Same as UTC
WET	00:00	Western European Time
WAT	-01:00	West Africa Time
FNST	-01:00	Fernando de Noronha Summer Time
FNT	-02:00	Fernando de Noronha Time
BRST	-02:00	Brasilia Summer Time
NDT	-02:30	Newfoundland Daylight-Saving Time
ADT	-03:00	Atlantic Daylight-Saving Time
AWT	-03:00	(unknown)
BRT	-03:00	Brasilia Time
NFT	-03:30	Newfoundland Standard Time
NST	-03:30	Newfoundland Standard Time
AST	-04:00	Atlantic Standard Time (Canada)
ACST	-04:00	Atlantic/Porto Acre Summer Time
EDT	-04:00	Eastern Daylight-Saving Time
ACT	-05:00	Atlantic/Porto Acre Standard Time
CDT	-05:00	Central Daylight-Saving Time
EST	-05:00	Eastern Standard Time
CST	-06:00	Central Standard Time
MDT	-06:00	Mountain Daylight-Saving Time
MST	-07:00	Mountain Standard Time
PDT	-07:00	Pacific Daylight-Saving Time



Zona horária	Deslocamento da UTC	Descrição
AKDT	-08:00	Alaska Daylight-Saving Time
PST	-08:00	Pacific Standard Time
YDT	-08:00	Yukon Daylight-Saving Time
AKST	-09:00	Alaska Standard Time
HDT	-09:00	Hawaii/Alaska Daylight-Saving Time
YST	-09:00	Yukon Standard Time
MART	-09:30	Marquesas Time
AHST	-10:00	Alaska/Hawaii Standard Time
HST	-10:00	Hawaii Standard Time
CAT	-10:00	Central Alaska Time
NT	-11:00	Nome Time
IDLW	-12:00	International Date Line, West

**Zonas horárias da Austrália.** Existem três nomes em conflito entre as zonas horárias da Austrália e as zonas horárias utilizadas normalmente na América do Sul e do Norte: ACST, CST e EST. Se a opção em tempo de execução `australian_timezones` estiver definida como verdade, então ACST, CST, EST e SAT são interpretados como nomes de zona horária da Austrália, conforme mostrado na Tabela B-5. Se estiver definido como falso (que é o padrão), então ACST, CST e EST são tomados como nomes de zona horária das Américas, e SAT é interpretado como palavra sem efeito indicando Sábado.

**Tabela B-5. Abreviaturas das zonas horárias da Austrália para entrada**

Zona horária	Deslocamento da UTC	Descrição
ACST	+09:30	Central Australia Standard Time
CST	+10:30	Australian Central Standard Time
EST	+10:00	Australian Eastern Standard Time
SAT	+09:30	South Australian Standard Time

A Tabela B-6 mostra os nomes de zona horária reconhecidos pelo PostgreSQL como definição válida do parâmetro `timezone`. Deve ser observado que estes nomes são conceitualmente semelhantes, mas na prática diferentes, dos nomes mostrados na Tabela B-4: a maior parte destes nomes implica em uma regra de horário de verão (`daylight-savings time rule`) local, enquanto os nomes da Tabela B-4 representam apenas um deslocamento fixo em relação à UTC.

Em muitos casos existem diversos nomes equivalentes para a mesma zona horária. Estes são listados na mesma linha. A tabela é ordenada, preferencialmente, pelo nome da principal cidade da zona horária.

**Tabela B-6. Nomes de zona horária para definir `timezone`**

Zona horária
Africa/Abidjan
Africa/Accra
Africa/Addis_Ababa
Africa/Algiers
Africa/Asmera

<b>Zona horária</b>
Africa/Bamako
Africa/Bangui
Africa/Banjul
Africa/Bissau
Africa/Blantyre
Africa/Brazzaville
Africa/Bujumbura
Africa/Cairo Egypt
Africa/Casablanca
Africa/Ceuta
Africa/Conakry
Africa/Dakar
Africa/Dar_es_Salaam
Africa/Djibouti
Africa/Douala
Africa/El_Aaiun
Africa/Freetown
Africa/Gaborone
Africa/Harare
Africa/Johannesburg
Africa/Kampala
Africa/Khartoum
Africa/Kigali
Africa/Kinshasa
Africa/Lagos
Africa/Libreville
Africa/Lome
Africa/Luanda
Africa/Lubumbashi
Africa/Lusaka
Africa/Malabo
Africa/Maputo
Africa/Maseru
Africa/Mbabane
Africa/Mogadishu

<b>Zona horária</b>
Africa/Monrovia
Africa/Nairobi
Africa/Ndjamena
Africa/Niamey
Africa/Nouakchott
Africa/Ouagadougou
Africa/Porto-Novo
Africa/Sao_Tome
Africa/Timbuktu
Africa/Tripoli Libya
Africa/Tunis
Africa/Windhoek
America/Adak America/Atka US/Aleutian
America/Anchorage SystemV/YST9YDT US/Alaska
America/Anguilla
America/Antigua
America/Araguaina
America/Aruba
America/Asuncion
America/Bahia
America/Barbados
America/Belem
America/Belize
America/Boa_Vista
America/Bogota
America/Boise
America/Buenos_Aires
America/Cambridge_Bay
America/Campo_Grande
America/Cancun
America/Caracas
America/Catamarca
America/Cayenne
America/Cayman
America/Chicago CST6CDT SystemV/CST6CDT US/Central

<b>Zona horária</b>
America/Chihuahua
America/Cordoba America/Rosario
America/Costa_Rica
America/Cuiaba
America/Curacao
America/Danmarkshavn
America/Dawson
America/Dawson_Creek
America/Denver MST7MDT SystemV/MST7MDT US/Mountain America/Shiprock Navajo
America/Detroit US/Michigan
America/Dominica
America/Edmonton Canada/Mountain
America/Eirunepe
America/El_Salvador
America/Ensenada America/Tijuana Mexico/BajaNorte
America/Fortaleza
America/Glace_Bay
America/Godthab
America/Goose_Bay
America/Grand_Turk
America/Grenada
America/Guadeloupe
America/Guatemala
America/Guayaquil
America/Guyana
America/Halifax Canada/Atlantic SystemV/AST4ADT
America/Havana Cuba
America/Hermosillo
America/Indiana/Indianapolis America/Indianapolis America/Fort_Wayne EST SystemV/EST5 US/East-Indiana
America/Indiana/Knox America/Knox_IN US/Indiana-Starke
America/Indiana/Marengo
America/Indiana/Vevay
America/Inuvik
America/Iqaluit
America/Jamaica Jamaica

<b>Zona horária</b>
America/Jujuy
America/Juneau
America/Kentucky/Louisville America/Louisville
America/Kentucky/Monticello
America/La_Paz
America/Lima
America/Los_Angeles PST8PDT SystemV/PST8PDT US/Pacific US/Pacific-New
America/Maceio
America/Managua
America/Manaus Brazil/West
America/Martinique
America/Mazatlan Mexico/BajaSur
America/Mendoza
America/Menominee
America/Merida
America/Mexico_City Mexico/General
America/Miquelon
America/Monterrey
America/Montevideo
America/Montreal
America/Montserrat
America/Nassau
America/New_York EST5EDT SystemV/EST5EDT US/Eastern
America/Nipigon
America/Nome
America/Noronha Brazil/DeNoronha
America/North_Dakota/Center
America/Panama
America/Pangnirtung
America/Paramaribo
America/Phoenix MST SystemV/MST7 US/Arizona
America/Port-au-Prince
America/Port_of_Spain
America/Porto_Acre America/Rio_Branco Brazil/Acre
America/Porto_Velho

<b>Zona horária</b>
America/Puerto_Rico SystemV/AST4
America/Rainy_River
America/Rankin_Inlet
America/Recife
America/Regina Canada/East-Saskatchewan Canada/Saskatchewan SystemV/CST6
America/Santiago Chile/Continental
America/Santo_Domingo
America/Sao_Paulo Brazil/East
America/Scoresbysund
America/St_Johns Canada/Newfoundland
America/St_Kitts
America/St_Lucia
America/St_Thomas America/Virgin
America/St_Vincent
America/Swift_Current
America/Tegucigalpa
America/Thule
America/Thunder_Bay
America/Toronto Canada/Eastern
America/Tortola
America/Vancouver Canada/Pacific
America/Whitehorse Canada/Yukon
America/Winnipeg Canada/Central
America/Yakutat
America/Yellowknife
Antarctica/Casey
Antarctica/Davis
Antarctica/DumontDUrville
Antarctica/Mawson
Antarctica/McMurdo Antarctica/South_Pole
Antarctica/Palmer
Antarctica/Rothera
Antarctica/Syowa
Antarctica/Vostok
Asia/Aden

<b>Zona horária</b>
Asia/Almaty
Asia/Amman
Asia/Anadyr
Asia/Aqtau
Asia/Aqtobe
Asia/Ashgabat Asia/Ashkhabad
Asia/Baghdad
Asia/Bahrain
Asia/Baku
Asia/Bangkok
Asia/Beirut
Asia/Bishkek
Asia/Brunei
Asia/Calcutta
Asia/Choibalsan
Asia/Chongqing Asia/Chungking
Asia/Colombo
Asia/Dacca Asia/Dhaka
Asia/Damascus
Asia/Dili
Asia/Dubai
Asia/Dushanbe
Asia/Gaza
Asia/Harbin
Asia/Hong_Kong Hongkong
Asia/Hovd
Asia/Irkutsk
Asia/Jakarta
Asia/Jayapura
Asia/Jerusalem Asia/Tel_Aviv Israel
Asia/Kabul
Asia/Kamchatka
Asia/Karachi
Asia/Kashgar
Asia/Katmandu

<b>Zona horária</b>
Asia/Krasnoyarsk
Asia/Kuala_Lumpur
Asia/Kuching
Asia/Kuwait
Asia/Macao Asia/Macau
Asia/Magadan
Asia/Makassar Asia/Ujung_Pandang
Asia/Manila
Asia/Muscat
Asia/Nicosia Europe/Nicosia
Asia/Novosibirsk
Asia/Omsk
Asia/Oral
Asia/Phnom_Penh
Asia/Pontianak
Asia/Pyongyang
Asia/Qatar
Asia/Qyzylorda
Asia/Rangoon
Asia/Riyadh
Asia/Riyadh87 Mideast/Riyadh87
Asia/Riyadh88 Mideast/Riyadh88
Asia/Riyadh89 Mideast/Riyadh89
Asia/Saigon
Asia/Sakhalin
Asia/Samarkand
Asia/Seoul ROK
Asia/Shanghai PRC
Asia/Singapore Singapore
Asia/Taipei ROC
Asia/Tashkent
Asia/Tbilisi
Asia/Tehran Iran
Asia/Thimbu Asia/Thimphu
Asia/Tokyo Japan



<b>Zona horária</b>
Asia/Ulaanbaatar Asia/Ulan_Bator
Asia/Urumqi
Asia/Vientiane
Asia/Vladivostok
Asia/Yakutsk
Asia/Yekaterinburg
Asia/Yerevan
Atlantic/Azores
Atlantic/Bermuda
Atlantic/Canary
Atlantic/Cape_Verde
Atlantic/Faeroe
Atlantic/Madeira
Atlantic/Reykjavik Iceland
Atlantic/South_Georgia
Atlantic/St_Helena
Atlantic/Stanley
Australia/ACT Australia/Canberra Australia/NSW Australia/Sydney
Australia/Adelaide Australia/South
Australia/Brisbane Australia/Queensland
Australia/Broken_Hill Australia/Yancowinna
Australia/Darwin Australia/North
Australia/Hobart Australia/Tasmania
Australia/LHI Australia/Lord_Howe
Australia/Lindeman
Australia/Melbourne Australia/Victoria
Australia/Perth Australia/West
CET
EET
Etc/GMT+1
Etc/GMT+2
Etc/GMT+3
Etc/GMT+4
Etc/GMT+5
Etc/GMT+6

<b>Zona horária</b>
Etc/GMT+7
Etc/GMT+8
Etc/GMT+9
Etc/GMT+10
Etc/GMT+11
Etc/GMT+12
Etc/GMT-1
Etc/GMT-2
Etc/GMT-3
Etc/GMT-4
Etc/GMT-5
Etc/GMT-6
Etc/GMT-7
Etc/GMT-8
Etc/GMT-9
Etc/GMT-10
Etc/GMT-11
Etc/GMT-12
Etc/GMT-13
Etc/GMT-14
Europe/Amsterdam
Europe/Andorra
Europe/Athens
Europe/Belfast
Europe/Belgrade Europe/Ljubljana Europe/Sarajevo Europe/Skopje Europe/Zagreb
Europe/Berlin
Europe/Brussels
Europe/Bucharest
Europe/Budapest
Europe/Chisinau Europe/Tiraspol
Europe/Copenhagen
Europe/Dublin Eire
Europe/Gibraltar
Europe/Helsinki
Europe/Istanbul Asia/Istanbul Turkey

<b>Zona horária</b>
Europe/Kaliningrad
Europe/Kiev
Europe/Lisbon Portugal
Europe/London GB GB-Eire
Europe/Luxembourg
Europe/Madrid
Europe/Malta
Europe/Minsk
Europe/Monaco
Europe/Moscow W-SU
Europe/Oslo Arctic/Longyearbyen Atlantic/Jan_Mayen
Europe/Paris
Europe/Prague Europe/Bratislava
Europe/Riga
Europe/Rome Europe/San_Marino Europe/Vatican
Europe/Samara
Europe/Simferopol
Europe/Sofia
Europe/Stockholm
Europe/Tallinn
Europe/Tirane
Europe/Uzhgorod
Europe/Vaduz
Europe/Vienna
Europe/Vilnius
Europe/Warsaw Poland
Europe/Zaporozhye
Europe/Zurich
Factory
GMT GMT+0 GMT-0 GMT0 Greenwich Etc/GMT Etc/GMT+0 Etc/GMT-0 Etc/GMT0 Etc/Greenwich
Indian/Antananarivo
Indian/Chagos
Indian/Christmas
Indian/Cocos
Indian/Comoro

<b>Zona horária</b>
Indian/Kerguelen
Indian/Mahe
Indian/Maldives
Indian/Mauritius
Indian/Mayotte
Indian/Reunion
MET
Pacific/Apia
Pacific/Auckland NZ
Pacific/Chatham NZ-CHAT
Pacific/Easter Chile/EasterIsland
Pacific/Efate
Pacific/Enderbury
Pacific/Fakaofu
Pacific/Fiji
Pacific/Funafuti
Pacific/Galapagos
Pacific/Gambier SystemV/YST9
Pacific/Guadalcanal
Pacific/Guam
Pacific/Honolulu HST SystemV/HST10 US/Hawaii
Pacific/Johnston
Pacific/Kiritimati
Pacific/Kosrae
Pacific/Kwajalein Kwajalein
Pacific/Majuro
Pacific/Marquesas
Pacific/Midway
Pacific/Nauru
Pacific/Niue
Pacific/Norfolk
Pacific/Noumea
Pacific/Pago_Pago Pacific/Samoa US/Samoa
Pacific/Palau
Pacific/Pitcairn SystemV/PST8

Zona horária
Pacific/Ponape
Pacific/Port_Moresby
Pacific/Rarotonga
Pacific/Saipan
Pacific/Tahiti
Pacific/Tarawa
Pacific/Tongatapu
Pacific/Truk
Pacific/Wake
Pacific/Wallis
Pacific/Yap
UCT Etc/UCT
UTC Universal Zulu Etc/UTC Etc/Universal Etc/Zulu
WET

Além dos nomes listados nesta tabela, o PostgreSQL também aceita nomes de zona horária na forma *STDdeslocamento*, ou *STDdeslocamentoDST*, onde *STD* é uma abreviatura de zona horária, *deslocamento* é um deslocamento numérico em horas à oeste da UTC, e *DST* é uma abreviatura opcional de zona de horário de verão, assumida como significando uma hora à frente do deslocamento fornecido. Por exemplo, se *EST5EDT* já não fosse um nome de zona horária reconhecido, este seria aceito e seria funcionalmente equivalente ao horário da Costa Leste dos EUA. Quando o nome da zona de horário de verão não está presente, é assumido como sendo para ser usado de acordo com as regras de zona horária dos EUA e, portanto, esta funcionalidade tem pouca utilidade fora dos EUA. Deve haver preocupação com relação ao fato que esta funcionalidade pode levar à aceitação de entrada sem sentido, uma vez que não é verificado se a abreviatura da zona horária possui um valor razoável. Por exemplo, *SET TIMEZONE TO FOOBAR0* funciona, deixando o sistema utilizando uma abreviatura bem peculiar para GMT.

### B.3. História das unidades

A Data Juliana foi inventada pelo estudioso francês Joseph Justus Scaliger (1540-1609) e, provavelmente, recebeu este nome devido ao pai do Scaliger, o estudioso italiano Julius Caesar Scaliger (1484-1558). Os astrônomos têm utilizado a Data Juliana para atribuir um número único para cada dia a partir de 1 de janeiro de 4713 AC. Esta é a tão falada Data Juliana (DJ). DJ 0 (zero) designa as 24 horas que vão do meio-dia UTC de 1 de janeiro de 4713 AC até o meio-dia UTC de 2 de janeiro de 4713 AC.

A “Data Juliana” é diferente do “Calendário Juliano”. O Calendário Juliano foi introduzido por Julius Caesar em 45 AC. Ficou em uso corrente até 1582, quando os países começaram a mudar para o Calendário Gregoriano. No Calendário Juliano, o ano tropical é aproximado como  $365 \frac{1}{4}$  dias = 365,25 dias. Isto ocasiona um erro de aproximadamente 1 dia a cada 128 anos.

O erro acumulado fez o Papa Gregório XIII reformar o calendário de acordo com as instruções do Concílio de Trento. No Calendário Gregoriano o ano tropical é aproximado como  $365 + \frac{97}{400}$  dias = 365,2425 dias. Portanto, leva aproximadamente 3.300 anos para o ano tropical se deslocar um dia em relação ao Calendário Gregoriano.

A aproximação  $365 + \frac{97}{400}$  é obtida colocando 97 anos bissextos a cada 400 anos, utilizando as seguintes regras:

Todo ano divisível por 4 é um ano bissexto.

Entretanto, todo ano divisível por 100 não é um ano bissexto.

Entretanto, todo ano divisível por 400 é um ano bissexto sempre.

Portanto, 1700, 1800, 1900, 2100 e 2200 não são anos bissextos. Porém, 1600, 2000 e 2400 são anos bissextos. Contrapondo, no antigo Calendário Juliano todos os anos divisíveis por 4 são bissextos.

A Bula Papal de fevereiro de 1582 decretou que deveriam ser retirados 10 dias de outubro de 1582, fazendo com que 15 de outubro viesse imediatamente após 4 de outubro. Foi respeitado na Itália, Polônia, Portugal e Espanha. Outros países católicos seguiram logo após, mas os países protestantes relutaram em mudar, e os países ortodoxos gregos não mudaram até o início do século 20. A reforma foi seguida pela Inglaterra e seus domínios (incluindo o que agora são os EUA) em 1752. Assim, 2 de setembro de 1752 foi seguido por 14 de setembro de 1752. Por isso, nos sistemas Unix, o programa `cal` produz a seguinte informação:

```
$ cal 9 1752
```

```
      setembro 1752
Do Se Te Qu Qu Se Sá
          1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

**Nota:** O padrão SQL declara que “Dentro da definição do ‘literal data/hora’, os ‘valores de data/hora’ são restritos pelas regras naturais para data e hora de acordo com o Calendário Gregoriano”. As datas entre 1752-09-03 e 1752-09-13, embora excluídas em alguns países pela Bula Papal, estão em conformidade com as “regras naturais” sendo, portanto, datas válidas.

Foram elaborados calendários diferentes em várias partes do mundo, muitos anteriores ao sistema Gregoriano. Por exemplo, o início do Calendário Chinês remonta ao século 14 AC. Diz a lenda que o Imperador Huangdi inventou o calendário em 2637 AC. A República Popular da China utiliza o Calendário Gregoriano para as finalidades civis. O Calendário Chinês é utilizado para determinar as festividades.

## Notas

1. São dois os Calendário Cristãos ainda em uso no mundo: O Calendário Juliano foi proposto por Sosígenes, astrônomo de Alexandria, e introduzido por Julio César em 45 AC. Foi usado pelas igrejas e países cristãos até o século XVI, quando começou a ser trocado pelo Calendário Gregoriano. Alguns países, como a Grécia e a Rússia, o usaram até o século passado. Ainda é usado por algumas Igrejas Ortodoxas, entre elas a Igreja Russa; O Calendário Gregoriano foi proposto por Aloysius Lilius, astrônomo de Nápoles, e adotado pelo Papa Gregório XIII, seguindo as instruções do Concílio de Trento (1545-1563). O decreto instituindo esse calendário foi publicado em 24 de fevereiro de 1582. - Calendários (<http://www.observatorio.ufmg.br/pas39.htm>) (N. do T.)
2. AD = Anno Domini ([http://en.wikipedia.org/wiki/Anno\\_Domini](http://en.wikipedia.org/wiki/Anno_Domini)) (N. do T.)

## Apêndice C. Palavras chave do SQL

A Tabela C-1 lista todos os termos (tokens) que são palavras chave no padrão SQL e no PostgreSQL 8.0.0. Podem ser obtidas informações suplementares na Seção 4.1.1.

O padrão SQL faz distinção entre palavras chave *reservadas* e *não reservadas*. De acordo com o padrão, as palavras chave reservadas são as únicas palavras chave reais; nunca são permitidas como identificadores. As palavras chave não reservadas somente possuem significado especial em determinados contextos; podem ser utilizadas como identificador em outros contextos. Em sua maior parte, as palavras chave não reservadas são, na verdade, nomes de tabelas e funções nativas especificadas pelo padrão SQL. Essencialmente, o conceito de palavra chave não reservada existe apenas para declarar a associação desta palavra com um significado predefinido em alguns contextos.

No analisador do PostgreSQL a vida é um pouco mais complicada. Existem várias classes diferentes de termos, indo desde aquelas que nunca podem ser utilizadas como identificador, até aquelas que não possuem nenhum status especial no analisador se comparado com um identificador comum (Geralmente, esta última é o caso das funções especificadas pelo padrão SQL). Mesmo as palavras chave reservadas não são totalmente reservadas no PostgreSQL, sendo possível utilizá-las como títulos de colunas (por exemplo, `SELECT 55 AS CHECK`, embora `CHECK` seja uma palavra chave reservada).

Na coluna PostgreSQL da Tabela C-1, são classificadas como “não reservadas” as palavras chave explicitamente reconhecidas pelo analisador, mas permitidas na maioria ou em todos os contextos onde um identificador é esperado. Existem algumas palavras chave não reservadas que não podem ser utilizadas como nome de função ou de tipo de dado, estando devidamente indicado (Em sua maioria, estas palavras representam funções nativas ou tipos de dado com sintaxe especial. A função ou o tipo ainda está disponível, mas não pode ser redefinido pelo usuário). Na coluna “reservadas” estão os termos permitidos apenas como títulos de coluna utilizando “AS” (e, talvez, em muito poucos outros contextos). Algumas palavras chave reservadas são permitidas como nome de função; isto também está indicado na tabela.

Como regra geral, se acontecerem erros indevidos do analisador em comandos contendo como identificador qualquer uma das palavras chave listadas, deve-se tentar colocar o identificador entre aspas para ver se o problema desaparece.

Antes de estudar a Tabela C-1, é importante compreender o fato de uma palavra chave não ser reservada no PostgreSQL não significa que a funcionalidade associada a esta palavra chave não está implementada. Inversamente, a presença de uma palavra chave não indica a existência da funcionalidade.

**Tabela C-1. Palavras chave do SQL**

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
A		não-reservada		
ABORT	não-reservada			
ABS		reservada	não-reservada	
ABSOLUTE	não-reservada	não-reservada	reservada	reservada
ACCESS	não-reservada			
ACTION	não-reservada	não-reservada	reservada	reservada
ADA		não-reservada	não-reservada	não-reservada
ADD	não-reservada	não-reservada	reservada	reservada
ADMIN		não-reservada	reservada	

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
AFTER	não-reservada	não-reservada	reservada	
AGGREGATE	não-reservada		reservada	
ALIAS			reservada	
ALL	reservada	reservada	reservada	reservada
ALLOCATE		reservada	reservada	reservada
ALSO	não-reservada			
ALTER	não-reservada	reservada	reservada	reservada
ALWAYS		não-reservada		
ANALYSE	reservada			
ANALYZE	reservada			
AND	reservada	reservada	reservada	reservada
ANY	reservada	reservada	reservada	reservada
ARE		reservada	reservada	reservada
ARRAY	reservada	reservada	reservada	
AS	reservada	reservada	reservada	reservada
ASC	reservada	não-reservada	reservada	reservada
ASENSITIVE		reservada	não-reservada	
ASSERTION	não-reservada	não-reservada	reservada	reservada
ASSIGNMENT	não-reservada	não-reservada	não-reservada	
ASYMMETRIC		reservada	não-reservada	
AT	não-reservada	reservada	reservada	reservada
ATOMIC		reservada	não-reservada	
ATTRIBUTE		não-reservada		
ATTRIBUTES		não-reservada		
AUTHORIZATION	reservada (pode ser função)	reservada	reservada	reservada
AVG		reservada	não-reservada	reservada
BACKWARD	não-reservada			
BEFORE	não-reservada	não-	reservada	



Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
		reservada		
BEGIN	não-reservada	reservada	reservada	reservada
BERNOULLI		não-reservada		
BETWEEN	reservada (pode ser função)	reservada	não-reservada	reservada
BIGINT	não-reservada (não pode ser função ou tipo)	reservada		
BINARY	reservada (pode ser função)	reservada	reservada	
BIT	não-reservada (não pode ser função ou tipo)		reservada	reservada
BITVAR			não-reservada	
BIT_LENGTH			não-reservada	reservada
BLOB		reservada	reservada	
BOOLEAN	não-reservada (não pode ser função ou tipo)	reservada	reservada	
BOTH	reservada	reservada	reservada	reservada
BREADTH		não-reservada	reservada	
BY	não-reservada	reservada	reservada	reservada
C		não-reservada	não-reservada	não-reservada
CACHE	não-reservada			
CALL		reservada	reservada	
CALLED	não-reservada	reservada	não-reservada	
CARDINALITY		reservada	não-reservada	
CASCADE	não-reservada	não-reservada	reservada	reservada
CASCADEED		reservada	reservada	reservada
CASE	reservada	reservada	reservada	reservada
CAST	reservada	reservada	reservada	reservada
CATALOG		não-reservada	reservada	reservada
CATALOG_NAME		não-reservada	não-reservada	não-reservada
CEIL		reservada		

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
CEILING		reservada		
CHAIN	não-reservada	não-reservada	não-reservada	
CHAR	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
CHARACTER	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
CHARACTERISTICS	não-reservada	não-reservada		
CHARACTERS		não-reservada		
CHARACTER_LENGTH		reservada	não-reservada	reservada
CHARACTER_SET_CATALOG		não-reservada	não-reservada	não-reservada
CHARACTER_SET_NAME		não-reservada	não-reservada	não-reservada
CHARACTER_SET_SCHEMA		não-reservada	não-reservada	não-reservada
CHAR_LENGTH		reservada	não-reservada	reservada
CHECK	reservada	reservada	reservada	reservada
CHECKED			não-reservada	
CHECKPOINT	não-reservada			
CLASS	não-reservada		reservada	
CLASS_ORIGIN		não-reservada	não-reservada	não-reservada
CLOB		reservada	reservada	
CLOSE	não-reservada	reservada	reservada	reservada
CLUSTER	não-reservada			
COALESCE	não-reservada (não pode ser função ou tipo)	reservada	não-reservada	reservada
COBOL		não-reservada	não-reservada	não-reservada
COLLATE	reservada	reservada	reservada	reservada
COLLATION		não-reservada	reservada	reservada
COLLATION_CATALOG		não-reservada	não-reservada	não-reservada

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
COLLATION_NAME		não-reservada	não-reservada	não-reservada
COLLATION_SCHEMA		não-reservada	não-reservada	não-reservada
COLLECT		reservada		
COLUMN	reservada	reservada	reservada	reservada
COLUMN_NAME		não-reservada	não-reservada	não-reservada
COMMAND_FUNCTION		não-reservada	não-reservada	não-reservada
COMMAND_FUNCTION_CODE		não-reservada	não-reservada	
COMMENT	não-reservada			
COMMIT	não-reservada	reservada	reservada	reservada
COMMITTED	não-reservada	não-reservada	não-reservada	não-reservada
COMPLETION			reservada	
CONDITION		reservada		
CONDITION_NUMBER		não-reservada	não-reservada	não-reservada
CONNECT		reservada	reservada	reservada
CONNECTION		não-reservada	reservada	reservada
CONNECTION_NAME		não-reservada	não-reservada	não-reservada
CONSTRAINT	reservada	reservada	reservada	reservada
CONSTRAINTS	não-reservada	não-reservada	reservada	reservada
CONSTRAINT_CATALOG		não-reservada	não-reservada	não-reservada
CONSTRAINT_NAME		não-reservada	não-reservada	não-reservada
CONSTRAINT_SCHEMA		não-reservada	não-reservada	não-reservada
CONSTRUCTOR		não-reservada	reservada	
CONTAINS		não-reservada	não-reservada	
CONTINUE		não-reservada	reservada	reservada

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
CONVERSION	não-reservada			
CONVERT	não-reservada (não pode ser função ou tipo)	reservada	não-reservada	reservada
COPY	não-reservada			
CORR		reservada		
CORRESPONDING		reservada	reservada	reservada
COUNT		reservada	não-reservada	reservada
COVAR_POP		reservada		
COVAR_SAMP		reservada		
CREATE	reservada	reservada	reservada	reservada
CREATEDB	não-reservada			
CREATEUSER	não-reservada			
CROSS	reservada (pode ser função)	reservada	reservada	reservada
CSV	não-reservada			
CUBE		reservada	reservada	
CUME_DIST		reservada		
CURRENT		reservada	reservada	reservada
CURRENT_DATE	reservada	reservada	reservada	reservada
CURRENT_DEFAULT_TRANSFORM_GROUP		reservada		
CURRENT_PATH		reservada	reservada	
CURRENT_ROLE		reservada	reservada	
CURRENT_TIME	reservada	reservada	reservada	reservada
CURRENT_TIMESTAMP	reservada	reservada	reservada	reservada
CURRENT_TRANSFORM_GROUP_FOR_TYPE		reservada		
CURRENT_USER	reservada	reservada	reservada	reservada
CURSOR	não-reservada	reservada	reservada	reservada
CURSOR_NAME		não-reservada	não-reservada	não-reservada
CYCLE	não-reservada	reservada	reservada	
DATA		não-reservada	reservada	não-reservada
DATABASE	não-reservada			
DATE		reservada	reservada	reservada
DATETIME_INTERVAL_CODE		não-reservada	não-reservada	não-reservada
DATETIME_INTERVAL_PRECISION		não-	não-	não-

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
		reservada	reservada	reservada
DAY	não-reservada	reservada	reservada	reservada
DEALLOCATE	não-reservada	reservada	reservada	reservada
DEC	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
DECIMAL	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
DECLARE	não-reservada	reservada	reservada	reservada
DEFAULT	reservada	reservada	reservada	reservada
DEFAULTS	não-reservada	não-reservada		
DEFERRABLE	reservada	não-reservada	reservada	reservada
DEFERRED	não-reservada	não-reservada	reservada	reservada
DEFINED		não-reservada	não-reservada	
DEFINER	não-reservada	não-reservada	não-reservada	
DEGREE		não-reservada		
DELETE	não-reservada	reservada	reservada	reservada
DELIMITER	não-reservada			
DELIMITERS	não-reservada			
DENSE_RANK		reservada		
DEPTH		não-reservada	reservada	
DEREF		reservada	reservada	
DERIVED		não-reservada		
DESC	reservada	não-reservada	reservada	reservada
DESCRIBE		reservada	reservada	reservada
DESCRIPTOR		não-reservada	reservada	reservada
DESTROY			reservada	
DESTRUCTOR			reservada	
DETERMINISTIC		reservada	reservada	
DIAGNOSTICS		não-reservada	reservada	reservada

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
DICTIONARY			reservada	
DISCONNECT		reservada	reservada	reservada
DISPATCH		não-reservada	não-reservada	
DISTINCT	reservada	reservada	reservada	reservada
DO	reservada			
DOMAIN	não-reservada	não-reservada	reservada	reservada
DOUBLE	não-reservada	reservada	reservada	reservada
DROP	não-reservada	reservada	reservada	reservada
DYNAMIC		reservada	reservada	
DYNAMIC_FUNCTION		não-reservada	não-reservada	não-reservada
DYNAMIC_FUNCTION_CODE		não-reservada	não-reservada	
EACH	não-reservada	reservada	reservada	
ELEMENT		reservada		
ELSE	reservada	reservada	reservada	reservada
ENCODING	não-reservada			
ENCRYPTED	não-reservada			
END	reservada	reservada	reservada	reservada
END-EXEC		reservada	reservada	reservada
EQUALS		não-reservada	reservada	
ESCAPE	não-reservada	reservada	reservada	reservada
EVERY		reservada	reservada	
EXCEPT	reservada	reservada	reservada	reservada
EXCEPTION		não-reservada	reservada	reservada
EXCLUDE		não-reservada		
EXCLUDING	não-reservada	não-reservada		
EXCLUSIVE	não-reservada			
EXEC		reservada	reservada	reservada
EXECUTE	não-reservada	reservada	reservada	reservada
EXISTING			não-reservada	

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
EXISTS	não-reservada (não pode ser função ou tipo)	reservada	não-reservada	reservada
EXP		reservada		
EXPLAIN	não-reservada			
EXTERNAL	não-reservada	reservada	reservada	reservada
EXTRACT	não-reservada (não pode ser função ou tipo)	reservada	não-reservada	reservada
FALSE	reservada	reservada	reservada	reservada
FETCH	não-reservada	reservada	reservada	reservada
FILTER		reservada		
FINAL		não-reservada	não-reservada	
FIRST	não-reservada	não-reservada	reservada	reservada
FLOAT	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
FLOOR		reservada		
FOLLOWING		não-reservada		
FOR	reservada	reservada	reservada	reservada
FORCE	não-reservada			
FOREIGN	reservada	reservada	reservada	reservada
FORTRAN		não-reservada	não-reservada	não-reservada
FORWARD	não-reservada			
FOUND		não-reservada	reservada	reservada
FREE		reservada	reservada	
FREEZE	reservada (pode ser função)			
FROM	reservada	reservada	reservada	reservada
FULL	reservada (pode ser função)	reservada	reservada	reservada
FUNCTION	não-reservada	reservada	reservada	
FUSION		reservada		
G		não-reservada	não-reservada	
GENERAL		não-reservada	reservada	
GENERATED		não-reservada	não-reservada	

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
GET		reservada	reservada	reservada
GLOBAL	não-reservada	reservada	reservada	reservada
GO		não-reservada	reservada	reservada
GOTO		não-reservada	reservada	reservada
GRANT	reservada	reservada	reservada	reservada
GRANTED		não-reservada	não-reservada	
GROUP	reservada	reservada	reservada	reservada
GROUPING		reservada	reservada	
HANDLER	não-reservada			
HAVING	reservada	reservada	reservada	reservada
HIERARCHY		não-reservada	não-reservada	
HOLD	não-reservada	reservada	não-reservada	
HOST			reservada	
HOURL	não-reservada	reservada	reservada	reservada
IDENTITY		reservada	reservada	reservada
IGNORE			reservada	
ILIKE	reservada (pode ser função)			
IMMEDIATE	não-reservada	não-reservada	reservada	reservada
IMMUTABLE	não-reservada			
IMPLEMENTATION		não-reservada	não-reservada	
IMPLICIT	não-reservada			
IN	reservada	reservada	reservada	reservada
INCLUDING	não-reservada	não-reservada		
INCREMENT	não-reservada	não-reservada		
INDEX	não-reservada			
INDICATOR		reservada	reservada	reservada
INFIX			não-reservada	
INHERITS	não-reservada			



Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
INITIALIZE			reservada	
INITIALLY	reservada	não-reservada	reservada	reservada
INNER	reservada (pode ser função)	reservada	reservada	reservada
INOUT	não-reservada (não pode ser função ou tipo)	reservada	reservada	
INPUT	não-reservada	não-reservada	reservada	reservada
INSENSITIVE	não-reservada	reservada	não-reservada	reservada
INSERT	não-reservada	reservada	reservada	reservada
INSTANCE		não-reservada	não-reservada	
INSTANTIABLE		não-reservada	não-reservada	
INSTEAD	não-reservada			
INT	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
INTEGER	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
INTERSECT	reservada	reservada	reservada	reservada
INTERSECTION		reservada		
INTERVAL	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
INTO	reservada	reservada	reservada	reservada
INVOKER	não-reservada	não-reservada	não-reservada	
IS	reservada (pode ser função)	reservada	reservada	reservada
ISNULL	reservada (pode ser função)			
ISOLATION	não-reservada	não-reservada	reservada	reservada
ITERATE			reservada	
JOIN	reservada (pode ser função)	reservada	reservada	reservada
K		não-reservada	não-reservada	
KEY	não-reservada	não-reservada	reservada	reservada
KEY_MEMBER		não-reservada	não-reservada	
KEY_TYPE		não-	não-	

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
		reservada	reservada	
LANCOMPILER	não-reservada			
LANGUAGE	não-reservada	reservada	reservada	reservada
LARGE	não-reservada	reservada	reservada	
LAST	não-reservada	não-reservada	reservada	reservada
LATERAL		reservada	reservada	
LEADING	reservada	reservada	reservada	reservada
LEFT	reservada (pode ser função)	reservada	reservada	reservada
LENGTH		não-reservada	não-reservada	não-reservada
LESS			reservada	
LEVEL	não-reservada	não-reservada	reservada	reservada
LIKE	reservada (pode ser função)	reservada	reservada	reservada
LIMIT	reservada		reservada	
LISTEN	não-reservada			
LN		reservada		
LOAD	não-reservada			
LOCAL	não-reservada	reservada	reservada	reservada
LOCALTIME	reservada	reservada	reservada	
LOCALTIMESTAMP	reservada	reservada	reservada	
LOCATION	não-reservada			
LOCATOR		não-reservada	reservada	
LOCK	não-reservada			
LOWER		reservada	não-reservada	reservada
M		não-reservada	não-reservada	
MAP		não-reservada	reservada	
MATCH	não-reservada	reservada	reservada	reservada
MATCHED		não-reservada		
MAX		reservada	não-reservada	reservada
MAXVALUE	não-reservada	não-reservada		

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
MEMBER		reservada		
MERGE		reservada		
MESSAGE_LENGTH		não-reservada	não-reservada	não-reservada
MESSAGE_OCTET_LENGTH		não-reservada	não-reservada	não-reservada
MESSAGE_TEXT		não-reservada	não-reservada	não-reservada
METHOD		reservada	não-reservada	
MIN		reservada	não-reservada	reservada
MINUTE	não-reservada	reservada	reservada	reservada
MINVALUE	não-reservada	não-reservada		
MOD		reservada	não-reservada	
MODE	não-reservada			
MODIFIES		reservada	reservada	
MODIFY			reservada	
MODULE		reservada	reservada	reservada
MONTH	não-reservada	reservada	reservada	reservada
MORE		não-reservada	não-reservada	não-reservada
MOVE	não-reservada			
MULTISET		reservada		
MUMPS		não-reservada	não-reservada	não-reservada
NAME		não-reservada	não-reservada	não-reservada
NAMES	não-reservada	não-reservada	reservada	reservada
NATIONAL	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
NATURAL	reservada (pode ser função)	reservada	reservada	reservada
NCHAR	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
NCLOB		reservada	reservada	
NESTING		não-reservada		

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
NEW	reservada	reservada	reservada	
NEXT	não-reservada	não-reservada	reservada	reservada
NO	não-reservada	reservada	reservada	reservada
NOCREATEDB	não-reservada			
NOCREATEUSER	não-reservada			
NONE	não-reservada (não pode ser função ou tipo)	reservada	reservada	
NORMALIZE		reservada		
NORMALIZED		não-reservada		
NOT	reservada	reservada	reservada	reservada
NOTHING	não-reservada			
NOTIFY	não-reservada			
NOTNULL	reservada (pode ser função)			
NOWAIT	não-reservada			
NULL	reservada	reservada	reservada	reservada
NULLABLE		não-reservada	não-reservada	não-reservada
NULLIF	não-reservada (não pode ser função ou tipo)	reservada	não-reservada	reservada
NULLS		não-reservada		
NUMBER		não-reservada	não-reservada	não-reservada
NUMERIC	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
OBJECT	não-reservada	não-reservada	reservada	
OCTETS		não-reservada		
OCTET_LENGTH		reservada	não-reservada	reservada
OF	não-reservada	reservada	reservada	reservada
OFF	reservada		reservada	
OFFSET	reservada			
OIDS	não-reservada			
OLD	reservada	reservada	reservada	
ON	reservada	reservada	reservada	reservada

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
ONLY	reservada	reservada	reservada	reservada
OPEN		reservada	reservada	reservada
OPERATION			reservada	
OPERATOR	não-reservada			
OPTION	não-reservada	não-reservada	reservada	reservada
OPTIONS		não-reservada	não-reservada	
OR	reservada	reservada	reservada	reservada
ORDER	reservada	reservada	reservada	reservada
ORDERING		não-reservada		
ORDINALITY		não-reservada	reservada	
OTHERS		não-reservada		
OUT	não-reservada (não pode ser função ou tipo)	reservada	reservada	
OUTER	reservada (pode ser função)	reservada	reservada	reservada
OUTPUT		não-reservada	reservada	reservada
OVER		reservada		
OVERLAPS	reservada (pode ser função)	reservada	não-reservada	reservada
OVERLAY	não-reservada (não pode ser função ou tipo)	reservada	não-reservada	
OVERRIDING		não-reservada	não-reservada	
OWNER	não-reservada			
PAD		não-reservada	reservada	reservada
PARAMETER		reservada	reservada	
PARAMETERS			reservada	
PARAMETER_MODE		não-reservada	não-reservada	
PARAMETER_NAME		não-reservada	não-reservada	
PARAMETER_ORDINAL_POSITION		não-reservada	não-reservada	
PARAMETER_SPECIFIC_CATALOG		não-	não-	

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
		reservada	reservada	
PARAMETER_SPECIFIC_NAME		não-reservada	não-reservada	
PARAMETER_SPECIFIC_SCHEMA		não-reservada	não-reservada	
PARTIAL	não-reservada	não-reservada	reservada	reservada
PARTITION		reservada		
PASCAL		não-reservada	não-reservada	não-reservada
PASSWORD	não-reservada			
PATH		não-reservada	reservada	
PERCENTILE_CONT		reservada		
PERCENTILE_DISC		reservada		
PERCENT_RANK		reservada		
PLACING	reservada	não-reservada		
PLI		não-reservada	não-reservada	não-reservada
POSITION	não-reservada (não pode ser função ou tipo)	reservada	não-reservada	reservada
POSTFIX			reservada	
POWER		reservada		
PRECEDING		não-reservada		
PRECISION	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
PREFIX			reservada	
PREORDER			reservada	
PREPARE	não-reservada	reservada	reservada	reservada
PRESERVE	não-reservada	não-reservada	reservada	reservada
PRIMARY	reservada	reservada	reservada	reservada
PRIOR	não-reservada	não-reservada	reservada	reservada
PRIVILEGES	não-reservada	não-reservada	reservada	reservada
PROCEDURAL	não-reservada			
PROCEDURE	não-reservada	reservada	reservada	reservada

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
PUBLIC		não-reservada	reservada	reservada
QUOTE	não-reservada			
RANGE		reservada		
RANK		reservada		
READ	não-reservada	não-reservada	reservada	reservada
READS		reservada	reservada	
REAL	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
RECHECK	não-reservada			
RECURSIVE		reservada	reservada	
REF		reservada	reservada	
REFERENCES	reservada	reservada	reservada	reservada
REFERENCING		reservada	reservada	
REGR_AVGX		reservada		
REGR_AVGY		reservada		
REGR_COUNT		reservada		
REGR_INTERCEPT		reservada		
REGR_R2		reservada		
REGR_SLOPE		reservada		
REGR_SXX		reservada		
REGR_SXY		reservada		
REGR_SYY		reservada		
REINDEX	não-reservada			
RELATIVE	não-reservada	não-reservada	reservada	reservada
RELEASE	não-reservada	reservada		
RENAME	não-reservada			
REPEATABLE	não-reservada	não-reservada	não-reservada	não-reservada
REPLACE	não-reservada			
RESET	não-reservada			
RESTART	não-reservada	não-reservada		
RESTRICT	não-reservada	não-reservada	reservada	reservada

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
RESULT		reservada	reservada	
RETURN		reservada	reservada	
RETURNED_CARDINALITY		não-reservada		
RETURNED_LENGTH		não-reservada	não-reservada	não-reservada
RETURNED_OCTET_LENGTH		não-reservada	não-reservada	não-reservada
RETURNED_SQLSTATE		não-reservada	não-reservada	não-reservada
RETURNS	não-reservada	reservada	reservada	
REVOKE	não-reservada	reservada	reservada	reservada
RIGHT	reservada (pode ser função)	reservada	reservada	reservada
ROLE		não-reservada	reservada	
ROLLBACK	não-reservada	reservada	reservada	reservada
ROLLUP		reservada	reservada	
ROUTINE		não-reservada	reservada	
ROUTINE_CATALOG		não-reservada	não-reservada	
ROUTINE_NAME		não-reservada	não-reservada	
ROUTINE_SCHEMA		não-reservada	não-reservada	
ROW	não-reservada (não pode ser função ou tipo)	reservada	reservada	
ROWS	não-reservada	reservada	reservada	reservada
ROW_COUNT		não-reservada	não-reservada	não-reservada
ROW_NUMBER		reservada		
RULE	não-reservada			
SAVEPOINT	não-reservada	reservada	reservada	
SCALE		não-reservada	não-reservada	não-reservada
SCHEMA	não-reservada	não-reservada	reservada	reservada
SCHEMA_NAME		não-reservada	não-reservada	não-reservada
SCOPE		reservada	reservada	



Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
SCOPE_CATALOG		não-reservada		
SCOPE_NAME		não-reservada		
SCOPE_SCHEMA		não-reservada		
SCROLL	não-reservada	reservada	reservada	reservada
SEARCH		reservada	reservada	
SECOND	não-reservada	reservada	reservada	reservada
SECTION		não-reservada	reservada	reservada
SECURITY	não-reservada	não-reservada	não-reservada	
SELECT	reservada	reservada	reservada	reservada
SELF		não-reservada	não-reservada	
SENSITIVE		reservada	não-reservada	
SEQUENCE	não-reservada	não-reservada	reservada	
SERIALIZABLE	não-reservada	não-reservada	não-reservada	não-reservada
SERVER_NAME		não-reservada	não-reservada	não-reservada
SESSION	não-reservada	não-reservada	reservada	reservada
SESSION_USER	reservada	reservada	reservada	reservada
SET	não-reservada	reservada	reservada	reservada
SETOF	não-reservada (não pode ser função ou tipo)			
SETS		não-reservada	reservada	
SHARE	não-reservada			
SHOW	não-reservada			
SIMILAR	reservada (pode ser função)	reservada	não-reservada	
SIMPLE	não-reservada	não-reservada	não-reservada	
SIZE		não-reservada	reservada	reservada

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
SMALLINT	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
SOME	reservada	reservada	reservada	reservada
SOURCE		não-reservada	não-reservada	
SPACE		não-reservada	reservada	reservada
SPECIFIC		reservada	reservada	
SPECIFICTYPE		reservada	reservada	
SPECIFIC_NAME		não-reservada	não-reservada	
SQL		reservada	reservada	reservada
SQLCODE				reservada
SQLERROR				reservada
SQLEXCEPTION		reservada	reservada	
SQLSTATE		reservada	reservada	reservada
SQLWARNING		reservada	reservada	
SQRT		reservada		
STABLE	não-reservada			
START	não-reservada	reservada	reservada	
STATE		não-reservada	reservada	
STATEMENT	não-reservada	não-reservada	reservada	
STATIC		reservada	reservada	
STATISTICS	não-reservada			
STDDEV_POP		reservada		
STDDEV_SAMP		reservada		
STDIN	não-reservada			
STDOUT	não-reservada			
STORAGE	não-reservada			
STRICT	não-reservada			
STRUCTURE		não-reservada	reservada	
STYLE		não-reservada	não-reservada	
SUBCLASS_ORIGIN		não-reservada	não-reservada	não-reservada

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
SUBLIST			não-reservada	
SUBMULTISET		reservada		
SUBSTRING	não-reservada (não pode ser função ou tipo)	reservada	não-reservada	reservada
SUM		reservada	não-reservada	reservada
SYMMETRIC		reservada	não-reservada	
SYSID	não-reservada			
SYSTEM		reservada	não-reservada	
SYSTEM_USER		reservada	reservada	reservada
TABLE	reservada	reservada	reservada	reservada
TABLESAMPLE		reservada		
TABLESPACE	não-reservada			
TABLE_NAME		não-reservada	não-reservada	não-reservada
TEMP	não-reservada			
TEMPLATE	não-reservada			
TEMPORARY	não-reservada	não-reservada	reservada	reservada
TERMINATE			reservada	
THAN			reservada	
THEN	reservada	reservada	reservada	reservada
TIES		não-reservada		
TIME	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
TIMESTAMP	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
TIMEZONE_HOUR		reservada	reservada	reservada
TIMEZONE_MINUTE		reservada	reservada	reservada
TO	reservada	reservada	reservada	reservada
TOAST	não-reservada			
TOP_LEVEL_COUNT		não-reservada		
TRAILING	reservada	reservada	reservada	reservada
TRANSACTION	não-reservada	não-	reservada	reservada

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
		reservada		
TRANSACTIONS_COMMITTED		não-reservada	não-reservada	
TRANSACTIONS_ROLLED_BACK		não-reservada	não-reservada	
TRANSACTION_ACTIVE		não-reservada	não-reservada	
TRANSFORM		não-reservada	não-reservada	
TRANSFORMS		não-reservada	não-reservada	
TRANSLATE		reservada	não-reservada	reservada
TRANSLATION		reservada	reservada	reservada
TREAT	não-reservada (não pode ser função ou tipo)	reservada	reservada	
TRIGGER	não-reservada	reservada	reservada	
TRIGGER_CATALOG		não-reservada	não-reservada	
TRIGGER_NAME		não-reservada	não-reservada	
TRIGGER_SCHEMA		não-reservada	não-reservada	
TRIM	não-reservada (não pode ser função ou tipo)	reservada	não-reservada	reservada
TRUE	reservada	reservada	reservada	reservada
TRUNCATE	não-reservada			
TRUSTED	não-reservada			
TYPE	não-reservada	não-reservada	não-reservada	não-reservada
UESCAPE		reservada		
UNBOUNDED		não-reservada		
UNCOMMITTED	não-reservada	não-reservada	não-reservada	não-reservada
UNDER		não-reservada	reservada	
UNENCRYPTED	não-reservada			
UNION	reservada	reservada	reservada	reservada
UNIQUE	reservada	reservada	reservada	reservada

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
UNKNOWN	não-reservada	reservada	reservada	reservada
UNLISTEN	não-reservada			
UNNAMED		não-reservada	não-reservada	não-reservada
UNNEST		reservada	reservada	
UNTIL	não-reservada			
UPDATE	não-reservada	reservada	reservada	reservada
UPPER		reservada	não-reservada	reservada
USAGE	não-reservada	não-reservada	reservada	reservada
USER	reservada	reservada	reservada	reservada
USER_DEFINED_TYPE_CATALOG		não-reservada	não-reservada	
USER_DEFINED_TYPE_CODE		não-reservada		
USER_DEFINED_TYPE_NAME		não-reservada	não-reservada	
USER_DEFINED_TYPE_SCHEMA		não-reservada	não-reservada	
USING	reservada	reservada	reservada	reservada
VACUUM	não-reservada			
VALID	não-reservada			
VALIDATOR	não-reservada			
VALUE		reservada	reservada	reservada
VALUES	não-reservada	reservada	reservada	reservada
VARCHAR	não-reservada (não pode ser função ou tipo)	reservada	reservada	reservada
VARIABLE			reservada	
VARYING	não-reservada	reservada	reservada	reservada
VAR_POP		reservada		
VAR_SAMP		reservada		
VERBOSE	reservada (pode ser função)			
VIEW	não-reservada	não-reservada	reservada	reservada
VOLATILE	não-reservada			
WHEN	reservada	reservada	reservada	reservada
WHENEVER		reservada	reservada	reservada

Palavra chave	PostgreSQL	SQL:2003	SQL:1999	SQL-92
WHERE	reservada	reservada	reservada	reservada
WIDTH_BUCKET		reservada		
WINDOW		reservada		
WITH	não-reservada	reservada	reservada	reservada
WITHIN		reservada		
WITHOUT	não-reservada	reservada	reservada	
WORK	não-reservada	não-reservada	reservada	reservada
WRITE	não-reservada	não-reservada	reservada	reservada
YEAR	não-reservada		reservada	reservada
ZONE	não-reservada	não-reservada	reservada	reservada

### Exemplo C-1. Uso de palavra chave reservada como nome de função

Neste exemplo é utilizada a palavra chave reservada BETWEEN como nome de uma função.<sup>1</sup>

```
=> CREATE OR REPLACE FUNCTION between(int, int, int) RETURNS boolean AS '
'>     SELECT $1 BETWEEN $2 AND $3;
'> ' LANGUAGE SQL STRICT;
```

```
=> SELECT between(5,4,6);
```

```
between
-----
t
(1 linha)
```

```
=> SELECT between(6,2,4);
```

```
between
-----
f
(1 linha)
```

```
=> SELECT 5 BETWEEN 5 AND 6 AS between;
```

```
between
-----
t
(1 linha)
```

## Notas

1. Exemplo escrito pelo tradutor, não fazendo parte do manual original.

## Apêndice D. Conformidade com o padrão SQL

Esta seção procura delinear até que ponto o PostgreSQL está em conformidade com o padrão SQL corrente. As informações que se seguem não são uma declaração completa da conformidade, mas apresentam os tópicos principais em um nível de detalhe que é tanto razoável quanto útil para os usuários.

O nome formal do padrão SQL é ISO/IEC 9075 “Database Language SQL”. A versão revisada do padrão é liberada de tempos em tempos; a mais recente surgiu no final de 2003; Esta versão é referida como ISO/IEC 9075:2003, ou simplesmente como SQL:2003. As versões anteriores a esta foram SQL:1999 e SQL-92. Cada versão substitui a versão anterior, portanto declarar que está em conformidade com uma versão anterior não tem mérito oficial. O desenvolvimento do PostgreSQL tenta manter a conformidade com a última versão oficial do padrão, quando esta conformidade não contradiz as funcionalidades tradicionais ou o senso comum. Durante a preparação do SQL:2003 o projeto PostgreSQL não esteve representado no Grupo de Trabalho ISO/IEC 9075, mesmo assim muitas funcionalidades requeridas pelo SQL:2003 já são suportadas, embora às vezes com uma sintaxe ligeiramente diferente ou através de uma função. Pode-se esperar movimentos em direção à conformidade nas próximas versões.

O SQL-92 definiu três conjuntos de funcionalidades para conformidade: Entrada, Intermediário e Completo. A maioria dos sistemas gerenciadores de banco de dados que se declaravam em conformidade com este padrão SQL, estavam em conformidade apenas com o nível de Entrada, porque o conjunto completo de funcionalidades nos níveis Intermediário e Completo era muito volumoso, ou conflitava com comportamentos legados.

A partir do SQL:1999, o padrão SQL passou a definir um conjunto maior de funcionalidades individuais, em vez dos três níveis grandes e ineficazes presentes no SQL-92. Um subconjunto grande destas funcionalidades representa as funcionalidades “núcleo” (*core*), que toda implementação em conformidade com o padrão SQL deve atender. As demais funcionalidades são inteiramente opcionais. Algumas funcionalidades opcionais são agrupadas para formarem “pacotes”, com os quais as implementações SQL podem se declarar em conformidade, portanto se declarando em conformidade com um determinado grupo de funcionalidades.

O padrão SQL:2003 também está dividido em partes. Cada uma destas partes é conhecida por um nome abreviado. Deve ser observado que estas partes não são numeradas sequencialmente.

- ISO/IEC 9075-1 Framework (SQL/Framework)
- ISO/IEC 9075-2 Foundation (SQL/Foundation)
- ISO/IEC 9075-3 Call Level Interface (SQL/CLI)
- ISO/IEC 9075-4 Persistent Stored Modules (SQL/PSM)
- ISO/IEC 9075-9 Management of External Data (SQL/MED)
- ISO/IEC 9075-10 Object Language Bindings (SQL/OLB)
- ISO/IEC 9075-11 Information and Definition Schemas (SQL/Schemata)
- ISO/IEC 9075-13 Routines and Types using the Java Language (SQL/JRT)
- ISO/IEC 9075-14 XML-related specifications (SQL/XML)

O PostgreSQL cobre as partes 1, 2 e 11. A parte 3 é semelhante à interface ODBC, e a parte 4 é semelhante à linguagem de programação PL/pgSQL, mas nestes casos não se busca ou verifica especificamente a conformidade exata.

O PostgreSQL suporta a maioria das funcionalidades principais do padrão SQL:2003. Das 164 funcionalidades requeridas para a conformidade total com o núcleo, o PostgreSQL está em conformidade com pelo menos 150. Além disso, existe uma longa lista de funcionalidades opcionais suportadas. Vale a pena notar que no momento em que esta documentação foi escrita nenhum sistema de gerenciamento de banco de dados se declarava em conformidade total com o núcleo do SQL:2003.

Nas seções que se seguem, é fornecida uma lista das funcionalidades que o PostgreSQL suporta, seguida por uma lista das funcionalidades definidas pelo SQL:2003 que o PostgreSQL ainda não suporta. As duas listas são aproximadas: podem existir pequenos detalhes que não estão em conformidade em uma funcionalidade declarada como suportada, enquanto grande parte de uma funcionalidade não suportada pode, na verdade, estar implementada. O corpo principal da documentação sempre contém informações mais precisas sobre o que funciona e o que não funciona.

**Nota:** Os códigos de funcionalidade contendo hífen são subfuncionalidades. Portanto, se uma determinada subfuncionalidade não for suportada, a funcionalidade principal é declarada como não suportada, mesmo que outras subfuncionalidades sejam suportadas.

## D.1. Funcionalidades suportadas

Identificador	Pacote	Descrição	Comentário
B012		Embedded C	
B021		Direct SQL	
E011	Core	Numeric data types	
E011-01	Core	INTEGER and SMALLINT data types	
E011-02	Core	REAL, DOUBLE PRECISION, and FLOAT data types	
E011-03	Core	DECIMAL and NUMERIC data types	
E011-04	Core	Arithmetic operators	
E011-05	Core	Numeric comparison	
E011-06	Core	Implicit casting among the numeric data types	
E021	Core	Character data types	
E021-01	Core	CHARACTER data type	
E021-02	Core	CHARACTER VARYING data type	
E021-03	Core	Character literals	
E021-04	Core	CHARACTER_LENGTH function	trims trailing spaces from CHARACTER values before counting
E021-05	Core	OCTET_LENGTH function	
E021-06	Core	SUBSTRING function	
E021-07	Core	Character concatenation	
E021-08	Core	UPPER and LOWER functions	
E021-09	Core	TRIM function	
E021-10	Core	Implicit casting among the character string types	
E021-11	Core	POSITION function	
E021-12	Core	Character comparison	
E031	Core	Identifiers	
E031-01	Core	Delimited identifiers	
E031-02	Core	Lower case identifiers	
E031-03	Core	Trailing underscore	
E051	Core	Basic query specification	
E051-01	Core	SELECT DISTINCT	



Identificador	Pacote	Descrição	Comentário
E051-02	Core	GROUP BY clause	
E051-04	Core	GROUP BY can contain columns not in <select list>	
E051-05	Core	Select list items can be renamed	AS is required
E051-06	Core	HAVING clause	
E051-07	Core	Qualified * in select list	
E051-08	Core	Correlation names in the FROM clause	
E051-09	Core	Rename columns in the FROM clause	
E061	Core	Basic predicates and search conditions	
E061-01	Core	Comparison predicate	
E061-02	Core	BETWEEN predicate	
E061-03	Core	IN predicate with list of values	
E061-04	Core	LIKE predicate	
E061-05	Core	LIKE predicate ESCAPE clause	
E061-06	Core	NULL predicate	
E061-07	Core	Quantified comparison predicate	
E061-08	Core	EXISTS predicate	
E061-09	Core	Subqueries in comparison predicate	
E061-11	Core	Subqueries in IN predicate	
E061-12	Core	Subqueries in quantified comparison predicate	
E061-13	Core	Correlated subqueries	
E061-14	Core	Search condition	
E071	Core	Basic query expressions	
E071-01	Core	UNION DISTINCT table operator	
E071-02	Core	UNION ALL table operator	
E071-03	Core	EXCEPT DISTINCT table operator	
E071-05	Core	Columns combined via table operators need not have exactly the same data type	
E071-06	Core	Table operators in subqueries	
E081-01	Core	SELECT privilege	
E081-02	Core	DELETE privilege	
E081-03	Core	INSERT privilege at the table level	
E081-04	Core	UPDATE privilege at the table level	
E081-06	Core	REFERENCES privilege at the table level	
E081-08	Core	WITH GRANT OPTION	
E081-10	Core	EXECUTE privilege	

Identificador	Pacote	Descrição	Comentário
E091	Core	Set functions	
E091-01	Core	AVG	
E091-02	Core	COUNT	
E091-03	Core	MAX	
E091-04	Core	MIN	
E091-05	Core	SUM	
E091-06	Core	ALL quantifier	
E091-07	Core	DISTINCT quantifier	
E101	Core	Basic data manipulation	
E101-01	Core	INSERT statement	
E101-03	Core	Searched UPDATE statement	
E101-04	Core	Searched DELETE statement	
E111	Core	Single row SELECT statement	
E121-01	Core	DECLARE CURSOR	
E121-02	Core	ORDER BY columns need not be in select list	
E121-03	Core	Value expressions in ORDER BY clause	
E121-04	Core	OPEN statement	
E121-08	Core	CLOSE statement	
E121-10	Core	FETCH statement implicit NEXT	
E121-17	Core	WITH HOLD cursors	
E131	Core	Null value support (nulls in lieu of values)	
E141	Core	Basic integrity constraints	
E141-01	Core	NOT NULL constraints	
E141-02	Core	UNIQUE constraints of NOT NULL columns	
E141-03	Core	PRIMARY KEY constraints	
E141-04	Core	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action	
E141-06	Core	CHECK constraints	
E141-07	Core	Column defaults	
E141-08	Core	NOT NULL inferred on PRIMARY KEY	
E141-10	Core	Names in a foreign key can be specified in any order	
E151	Core	Transaction support	
E151-01	Core	COMMIT statement	
E151-02	Core	ROLLBACK statement	
E152	Core	Basic SET TRANSACTION statement	

Identificador	Pacote	Descrição	Comentário
E152-01	Core	SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause	
E152-02	Core	SET TRANSACTION statement: READ ONLY and READ WRITE clauses	
E161	Core	SQL comments using leading double minus	
E171	Core	SQLSTATE support	
F021	Core	Basic information schema	
F021-01	Core	COLUMNS view	
F021-02	Core	TABLES view	
F021-03	Core	VIEWS view	
F021-04	Core	TABLE_CONSTRAINTS view	
F021-05	Core	REFERENTIAL_CONSTRAINTS view	
F021-06	Core	CHECK_CONSTRAINTS view	
F031	Core	Basic schema manipulation	
F031-01	Core	CREATE TABLE statement to create persistent base tables	
F031-02	Core	CREATE VIEW statement	
F031-03	Core	GRANT statement	
F031-04	Core	ALTER TABLE statement: ADD COLUMN clause	
F031-13	Core	DROP TABLE statement: RESTRICT clause	
F031-16	Core	DROP VIEW statement: RESTRICT clause	
F031-19	Core	REVOKE statement: RESTRICT clause	
F032		CASCADE drop behavior	
F033		ALTER TABLE statement: DROP COLUMN clause	
F034		Extended REVOKE statement	
F034-01		REVOKE statement performed by other than the owner of a schema object	
F034-02		REVOKE statement: GRANT OPTION FOR clause	
F034-03		REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION	
F041	Core	Basic joined table	
F041-01	Core	Inner join (but not necessarily the INNER keyword)	
F041-02	Core	INNER keyword	
F041-03	Core	LEFT OUTER JOIN	
F041-04	Core	RIGHT OUTER JOIN	
F041-05	Core	Outer joins can be nested	
F041-07	Core	The inner table in a left or right outer join can also be	

Identificador	Pacote	Descrição	Comentário
		used in an inner join	
F041-08	Core	All comparison operators are supported (rather than just =)	
F051	Core	Basic date and time	
F051-01	Core	DATE data type (including support of DATE literal)	
F051-02	Core	TIME data type (including support of TIME literal) with fractional seconds precision of at least 0	
F051-03	Core	TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6	
F051-04	Core	Comparison predicate on DATE, TIME, and TIMESTAMP data types	
F051-05	Core	Explicit CAST between datetime types and character string types	
F051-06	Core	CURRENT_DATE	
F051-07	Core	LOCALTIME	
F051-08	Core	LOCALTIMESTAMP	
F052	Enhanced datetime facilities	Intervals and datetime arithmetic	
F053		OVERLAPS predicate	
F081	Core	UNION and EXCEPT in views	
F111		Isolation levels other than SERIALIZABLE	
F111-01		READ UNCOMMITTED isolation level	
F111-02		READ COMMITTED isolation level	
F111-03		REPEATABLE READ isolation level	
F131	Core	Grouped operations	
F131-01	Core	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	
F131-02	Core	Multiple tables supported in queries with grouped views	
F131-03	Core	Set functions supported in queries with grouped views	
F131-04	Core	Subqueries with GROUP BY and HAVING clauses and grouped views	
F131-05	Core	Single row SELECT with GROUP BY and HAVING clauses and grouped views	
F171		Multiple schemas per user	
F191	Enhanced integrity management	Referential delete actions	
F201	Core	CAST function	

Identificador	Pacote	Descrição	Comentário
F221	Core	Explicit defaults	
F222		INSERT statement: DEFAULT VALUES clause	
F231		Privilege tables	
F231-01		TABLE_PRIVILEGES view	
F231-02		COLUMN_PRIVILEGES view	
F231-03		USAGE_PRIVILEGES view	
F251		Domain support	
F261	Core	CASE expression	
F261-01	Core	Simple CASE	
F261-02	Core	Searched CASE	
F261-03	Core	NULLIF	
F261-04	Core	COALESCE	
F271		Compound character literals	
F281		LIKE enhancements	
F302		INTERSECT table operator	
F302-01		INTERSECT DISTINCT table operator	
F302-02		INTERSECT ALL table operator	
F304		EXCEPT ALL table operator	
F311-01	Core	CREATE SCHEMA	
F311-02	Core	CREATE TABLE for persistent base tables	
F311-03	Core	CREATE VIEW	
F311-05	Core	GRANT statement	
F321		User authorization	
F361		Subprogram support	
F381		Extended schema manipulation	
F381-01		ALTER TABLE statement: ALTER COLUMN clause	
F381-02		ALTER TABLE statement: ADD CONSTRAINT clause	
F381-03		ALTER TABLE statement: DROP CONSTRAINT clause	
F391		Long identifiers	
F401		Extended joined table	
F401-01		NATURAL JOIN	
F401-02		FULL OUTER JOIN	
F401-04		CROSS JOIN	

Identificador	Pacote	Descrição	Comentário
F411	Enhanced datetime facilities	Time zone specification	differences regarding literal interpretation
F421		National character	
F431		Read-only scrollable cursors	
F431-01		FETCH with explicit NEXT	
F431-02		FETCH FIRST	
F431-03		FETCH LAST	
F431-04		FETCH PRIOR	
F431-05		FETCH ABSOLUTE	
F431-06		FETCH RELATIVE	
F441		Extended set function support	
F471	Core	Scalar subquery values	
F481	Core	Expanded NULL predicate	
F491	Enhanced integrity management	Constraint management	
F501	Core	Features and conformance views	
F501-01	Core	SQL_FEATURES view	
F501-02	Core	SQL_SIZING view	
F501-03	Core	SQL_LANGUAGES view	
F502		Enhanced documentation tables	
F502-01		SQL_SIZING_PROFILES view	
F502-02		SQL_IMPLEMENTATION_INFO view	
F502-03		SQL_PACKAGES view	
F531		Temporary tables	
F555	Enhanced datetime facilities	Enhanced seconds precision	
F561		Full value expressions	
F571		Truth value tests	
F591		Derived tables	
F611		Indicator data types	
F651		Catalog name qualifiers	
F672		Retrospective check constraints	
F701	Enhanced integrity management	Referential update actions	
F711		ALTER domain	
F761		Session management	

Identificador	Pacote	Descrição	Comentário
F771		Connection management	
F781		Self-referencing operations	
F791		Insensitive cursors	
F801		Full set function	
S071	Enhanced object support	SQL paths in function and type name resolution	
S111	Enhanced object support	ONLY in query expressions	
S211	Enhanced object support	User-defined cast functions	
T031		BOOLEAN data type	
T071		BIGINT data type	
T141		SIMILAR predicate	
T151		DISTINCT predicate	
T171		LIKE clause in table definition	
T191	Enhanced integrity management	Referential action RESTRICT	
T201	Enhanced integrity management	Comparable data types for referential constraints	
T211-01	Active database, Enhanced integrity management	Triggers activated on UPDATE, INSERT, or DELETE of one base table	
T211-02	Active database, Enhanced integrity management	BEFORE triggers	
T211-03	Active database, Enhanced integrity management	AFTER triggers	
T211-04	Active database, Enhanced integrity management	FOR EACH ROW triggers	
T211-07	Active database, Enhanced integrity management	TRIGGER privilege	
T212	Enhanced integrity management	Enhanced trigger capability	
T231		Sensitive cursors	
T241		START TRANSACTION statement	
T271		Savepoints	
T312		OVERLAY function	

Identificador	Pacote	Descrição	Comentário
T321-01	Core	User-defined functions with no overloading	
T321-03	Core	Function invocation	
T321-06	Core	ROUTINES view	
T321-07	Core	PARAMETERS view	
T322	PSM	Overloading of SQL-invoked functions and procedures	
T323		Explicit security for external routines	
T351		Bracketed SQL comments (/* ... */ comments)	
T441		ABS and MOD functions	
T501		Enhanced EXISTS predicate	
T551		Optional key words for default syntax	
T581		Regular expression substring function	
T591		UNIQUE constraints of possibly null columns	

## D.2. Funcionalidades não suportadas

As seguintes funcionalidades definidas no SQL:2003 não estão implementadas nesta versão do PostgreSQL. Em alguns poucos casos, está disponível uma funcionalidade equivalente.

Identificador	Pacote	Descrição	Comentário
B011		Embedded Ada	
B013		Embedded COBOL	
B014		Embedded Fortran	
B015		Embedded MUMPS	
B016		Embedded Pascal	
B017		Embedded PL/I	
B031		Basic dynamic SQL	
B032		Extended dynamic SQL	
B032-01		<describe input statement>	
B033		Untyped SQL-invoked function arguments	
B034		Dynamic specification of cursor attributes	
B041		Extensions to embedded SQL exception declarations	
B051		Enhanced execution rights	
B111		Module language Ada	
B112		Module language C	
B113		Module language COBOL	
B114		Module language Fortran	
B115		Module language MUMPS	



Identificador	Pacote	Descrição	Comentário
B116		Module language Pascal	
B117		Module language PL/I	
B121		Routine language Ada	
B122		Routine language C	
B123		Routine language COBOL	
B124		Routine language Fortran	
B125		Routine language MUMPS	
B126		Routine language Pascal	
B127		Routine language PL/I	
B128		Routine language SQL	
C011	Core	Call-Level Interface	
E081	Core	Basic Privileges	
E081-05	Core	UPDATE privilege at the column level	
E081-07	Core	REFERENCES privilege at the column level	
E081-09	Core	USAGE privilege	
E121	Core	Basic cursor support	
E121-06	Core	Positioned UPDATE statement	
E121-07	Core	Positioned DELETE statement	
E153	Core	Updatable queries with subqueries	
E182	Core	Module language	
F121		Basic diagnostics management	
F121-01		GET DIAGNOSTICS statement	
F121-02		SET TRANSACTION statement: DIAGNOSTICS SIZE clause	
F181	Core	Multiple module support	
F262		Extended CASE expression	
F263		Comma-separated predicates in simple CASE expression	
F291		UNIQUE predicate	
F301		CORRESPONDING in query expressions	
F311	Core	Schema definition statement	
F311-04	Core	CREATE VIEW: WITH CHECK OPTION	
F312		MERGE statement	
F341		Usage tables	
F392		Unicode escapes in identifiers	
F393		Unicode escapes in literals	

Identificador	Pacote	Descrição	Comentário
F402		Named column joins for LOBs, arrays, and multisets	
F442		Mixed column references in set functions	
F451		Character set definition	
F461		Named character sets	
F521	Enhanced integrity management	Assertions	
F641		Row and table constructors	
F661		Simple tables	
F671	Enhanced integrity management	Subqueries in CHECK	intentionally omitted
F691		Collation and translation	
F692		Enhanced collation support	
F693		SQL-session and client module collations	
F695		Translation support	
F696		Additional translation documentation	
F721		Deferrable constraints	foreign keys only
F731		INSERT column privileges	
F741		Referential MATCH types	no partial match yet
F751		View CHECK enhancements	
F811		Extended flagging	
F812	Core	Basic flagging	
F813		Extended flagging	
F821		Local table references	
F831		Full cursor update	
F831-01		Updatable scrollable cursors	
F831-02		Updatable ordered cursors	
S011	Core	Distinct data types	
S011-01	Core	USER_DEFINED_TYPES view	
S023	Basic object support	Basic structured types	
S024	Enhanced object support	Enhanced structured types	
S025		Final structured types	
S026		Self-referencing structured types	
S027		Create method by specific method name	
S028		Permutable UDT options list	

Identificador	Pacote	Descrição	Comentário
S041	Basic object support	Basic reference types	
S043	Enhanced object support	Enhanced reference types	
S051	Basic object support	Create table of type	
S081	Enhanced object support	Subtables	
S091		Basic array support	
S091-01		Arrays of built-in data types	
S091-02		Arrays of distinct types	
S091-03		Array expressions	
S092		Arrays of user-defined types	
S094		Arrays of reference types	
S095		Array constructors by query	
S096		Optional array bounds	
S097		Array element assignment	
S151	Basic object support	Type predicate	
S161	Enhanced object support	Subtype treatment	
S162		Subtype treatment for references	
S201		SQL-invoked routines on arrays	
S201-01		Array parameters	
S201-02		Array as result type of functions	
S202		SQL-invoked routines on multisets	
S231	Enhanced object support	Structured type locators	
S232		Array locators	
S233		Multiset locators	
S241		Transform functions	
S242		Alter transform statement	
S251		User-defined orderings	
S261		Specific type method	
S271		Basic multiset support	
S272		Multisets of user-defined types	
S274		Multisets of reference types	
S275		Advanced multiset support	
S281		Nested collection types	
S291		Unique constraint on entire row	
T011		Timestamp in Information Schema	
T041	Basic object support	Basic LOB data type support	

Identificador	Pacote	Descrição	Comentário
T041-01	Basic object support	BLOB data type	
T041-02	Basic object support	CLOB data type	
T041-03	Basic object support	POSITION, LENGTH, LOWER, TRIM, UPPER, and SUBSTRING functions for LOB data types	
T041-04	Basic object support	Concatenation of LOB data types	
T041-05	Basic object support	LOB locator: non-holdable	
T042		Extended LOB data type support	
T051		Row types	
T052		MAX and MIN for row types	
T053		Explicit aliases for all-fields reference	
T061		UCS support	
T111		Updatable joins, unions, and columns	
T121		WITH (excluding RECURSIVE) in query expression	
T122		WITH (excluding RECURSIVE) in subquery	
T131		Recursive query	
T132		Recursive query in subquery	
T152		DISTINCT predicate with negation	
T172		AS subquery clause in table definition	
T173		Extended LIKE clause in table definition	
T174		Identity columns	
T175		Generated columns	
T176		Sequence generator support	
T211	Active database, Enhanced integrity management	Basic trigger capability	
T211-05	Active database, Enhanced integrity management	Ability to specify a search condition that must be true before the trigger is invoked	
T211-06	Active database, Enhanced integrity management	Support for run-time rules for the interaction of triggers and constraints	
T211-08	Active database, Enhanced integrity management	Multiple triggers for the same event are executed in the order in which they were created in the catalog	intentionally omitted
T251		SET TRANSACTION statement: LOCAL option	
T261		Chained transactions	
T272		Enhanced savepoint management	
T281		SELECT privilege with column granularity	

Identificador	Pacote	Descrição	Comentário
T301		Functional dependencies	
T321	Core	Basic SQL-invoked routines	
T321-02	Core	User-defined stored procedures with no overloading	
T321-04	Core	CALL statement	
T321-05	Core	RETURN statement	
T324		Explicit security for SQL routines	
T325		Qualified SQL parameter references	
T326		Table functions	
T331		Basic roles	
T332		Extended roles	
T401		INSERT into a cursor	
T411		UPDATE statement: SET ROW option	
T431	OLAP	Extended grouping capabilities	
T432		Nested and concatenated GROUPING SETS	
T433		Multiargument GROUPING function	
T434		GROUP BY DISINCT	
T461		Symmetric BETWEEN predicate	
T471		Result sets return value	
T491		LATERAL derived table	
T511		Transaction counts	
T541		Updatable table references	
T561		Holdable locators	
T571		Array-returning external SQL-invoked functions	
T572		Multiset-returning external SQL-invoked functions	
T601		Local cursor references	
T611	OLAP	Elementary OLAP operations	
T612		Advanced OLAP operations	
T613		Sampling	
T621		Enhanced numeric functions	
T631	Core	IN predicate with one list element	
T641		Multiple column assignment	
T651		SQL-schema statements in SQL routines	
T652		SQL-dynamic statements in SQL routines	
T653		SQL-schema statements in external routines	
T654		SQL-dynamic statements in external routines	

Identificador	Pacote	Descrição	Comentário
T655		Cyclically dependent routines	

# Apêndice E. Release Notes

## E.1. Release 8.0

**Release date:** 2005-01-19

### E.1.1. Visão geral

Major changes in this release:

Microsoft Windows Native Server

This is the first PostgreSQL release to run natively on Microsoft Windows® as a server. It can run as a Windows service. This release supports NT-based Windows releases like Windows 2000, Windows XP, and Windows 2003. Older releases like Windows 95, Windows 98, and Windows ME are not supported because these operating systems do not have the infrastructure to support PostgreSQL. A separate installer project has been created to ease installation on Windows — see <http://pgfoundry.org/projects/pginstaller> (<http://pgfoundry.org/projects/pginstaller>).

Although tested throughout our release cycle, the Windows port does not have the benefit of years of use in production environments that PostgreSQL has on Unix platforms. Therefore it should be treated with the same level of caution as you would a new product.

Previous releases required the Unix emulation toolkit Cygwin in order to run the server on Windows operating systems. PostgreSQL has supported native clients on Windows for many years.

Savepoints

Savepoints allow specific parts of a transaction to be aborted without affecting the remainder of the transaction. Prior releases had no such capability; there was no way to recover from a statement failure within a transaction except by aborting the whole transaction. This feature is valuable for application writers who require error recovery within a complex transaction.

Point-In-Time Recovery

In previous releases there was no way to recover from disk drive failure except to restore from a previous backup or use a standby replication server. Point-in-time recovery allows continuous backup of the server. You can recover either to the point of failure or to some transaction in the past.

Tablespaces

Tablespaces allow administrators to select different file systems for storage of individual tables, indexes, and databases. This improves performance and control over disk space usage. Prior releases used `initlocation` and manual symlink management for such tasks.

Improved Buffer Management, `CHECKPOINT`, `VACUUM`

This release has a more intelligent buffer replacement strategy, which will make better use of available shared buffers and improve performance. The performance impact of vacuum and checkpoints is also lessened.

Change Column Types

A column's data type can now be changed with `ALTER TABLE`.

New Perl Server-Side Language

A new version of the `plperl` server-side language now supports a persistent shared storage area, triggers, returning records and arrays of records, and SPI calls to access the database.

Comma-separated-value (CSV) support in `COPY`

`COPY` can now read and write comma-separated-value files. It has the flexibility to interpret non-standard quoting and separation characters too.

## E.1.2. Migration to version 8.0

A dump/restore using `pg_dump` is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

- In `READ COMMITTED` serialization mode, volatile functions now see the results of concurrent transactions committed up to the beginning of each statement within the function, rather than up to the beginning of the interactive command that called the function.
- Functions declared `STABLE` or `IMMUTABLE` always use the snapshot of the calling query, and therefore do not see the effects of actions taken after the calling query starts, whether in their own transaction or other transactions. Such a function must be read-only, too, meaning that it cannot use any SQL commands other than `SELECT`.
- Non-deferred `AFTER` triggers are now fired immediately after completion of the triggering query, rather than upon finishing the current interactive command. This makes a difference when the triggering query occurred within a function: the trigger is invoked before the function proceeds to its next operation.
- Server configuration parameters `virtual_host` and `tcpip_socket` have been replaced with a more general parameter `listen_addresses`. Also, the server now listens on `localhost` by default, which eliminates the need for the `-i` postmaster switch in many scenarios.
- Server configuration parameters `SortMem` and `VacuumMem` have been renamed to `work_mem` and `maintenance_work_mem` to better reflect their use. The original names are still supported in `SET` and `SHOW`.
- Server configuration parameters `log_pid`, `log_timestamp`, and `log_source_port` have been replaced with a more general parameter `log_line_prefix`.
- Server configuration parameter `syslog` has been replaced with a more logical `log_destination` variable to control the log output destination.
- Server configuration parameter `log_statement` has been changed so it can selectively log just database modification or data definition statements. Server configuration parameter `log_duration` now prints only when `log_statement` prints the query.
- Server configuration parameter `max_expr_depth` parameter has been replaced with `max_stack_depth` which measures the physical stack size rather than the expression nesting depth. This helps prevent session termination due to stack overflow caused by recursive functions.
- The `length()` function no longer counts trailing spaces in `CHAR(n)` values.
- Casting an integer to `BIT(N)` selects the rightmost `N` bits of the integer, not the leftmost `N` bits as before.
- Updating an element or slice of a `NULL` array value now produces a non-`NULL` array result, namely an array containing just the assigned-to positions.
- Syntax checking of array input values has been tightened up considerably. Junk that was previously allowed in odd places with odd results now causes an error. Empty-string element values must now be written as `" "`, rather than writing nothing. Also changed behavior with respect to whitespace surrounding array elements: trailing whitespace is now ignored, for symmetry with leading whitespace (which has always been ignored).
- Overflow in integer arithmetic operations is now detected and reported as an error.
- The arithmetic operators associated with the single-byte `"char"` data type have been removed.
- The `extract()` function (also called `date_part`) now returns the proper year for BC dates. It previously returned one less than the correct year. The function now also returns the proper values for millennium and century.
- CIDR values now must have their non-masked bits be zero. For example, we no longer allow `204.248.199.1/31` as a CIDR value. Such values should never have been accepted by PostgreSQL and will now be rejected.
- `EXECUTE` now returns a completion tag that matches the executed statement.
- `psql`'s `\copy` command now reads or writes to the query's `stdin/stdout`, rather than `psql`'s `stdin/stdout`. The previous behavior can be accessed via new `pstdin/pstdout` parameters.
- The JDBC client interface has been removed from the core distribution, and is now hosted at <http://jdbc.postgresql.org>.



- The Tcl client interface has also been removed. There are several Tcl interfaces now hosted at <http://gborg.postgresql.org>.
- The server now uses its own time zone database, rather than the one supplied by the operating system. This will provide consistent behavior across all platforms. In most cases, there should be little noticeable difference in time zone behavior, except that the time zone names used by `SET/SHOW TimeZone` may be different from what your platform provides.
- Configure's threading option no longer requires users to run tests or edit configuration files; threading options are now detected automatically.
- Now that tablespaces have been implemented, `initlocation` has been removed.

### E.1.3. Deprecated Features

Some aspects of PostgreSQL's behavior have been determined to be suboptimal. For the sake of backward compatibility these have not been removed in 8.0, but they are considered deprecated and will be removed in the next major release.

- The 8.1 release will remove the function `to_char(interval, text)`.
- The server now warns of empty strings passed to `oid/float4/float8` data types, but continues to interpret them as zeroes as before. In the next major release, empty strings will be considered invalid input for these data types.
- By default, tables in PostgreSQL 8.0 and earlier are created with `OIDS`. In the next release, this will *not* be the case: to create a table that contains `OIDS`, the `WITH OIDS` clause must be specified or the `default_with_oids` configuration parameter must be set. Users are encouraged to explicitly specify `WITH OIDS` if their tables require `OIDS` for compatibility with future releases of PostgreSQL.

### E.1.4. Modificações

Below you will find a detailed account of the changes between release 8.0 and the previous major release.

#### E.1.4.1. Performance Improvements

- Support cross-data-type index usage (Tom)
 

Before this change, many queries would not use an index if the data types did not match exactly. This improvement makes index usage more intuitive and consistent.
- New buffer replacement strategy that improves caching (Jan)
 

Prior releases used a least-recently-used (LRU) cache to keep recently referenced pages in memory. The LRU algorithm did not consider the number of times a specific cache entry was accessed, so large table scans could force out useful cache pages. The new cache algorithm uses four separate lists to track most recently used and most frequently used cache pages and dynamically optimize their replacement based on the work load. This should lead to much more efficient use of the shared buffer cache. Administrators who have tested shared buffer sizes in the past should retest with this new cache replacement policy.
- Add subprocess to write dirty buffers periodically to reduce checkpoint writes (Jan)
 

In previous releases, the checkpoint process, which runs every few minutes, would write all dirty buffers to the operating system's buffer cache then flush all dirty operating system buffers to disk. This resulted in a periodic spike in disk usage that often hurt performance. The new code uses a background writer to trickle disk writes at a steady pace so checkpoints have far fewer dirty pages to write to disk. Also, the new code does not issue a global `sync()` call, but instead `fsync()`s just the files written since the last checkpoint. This should improve performance and minimize degradation during checkpoints.
- Add ability to prolong vacuum to reduce performance impact (Jan)
 

On busy systems, `VACUUM` performs many I/O requests which can hurt performance for other users. This release allows you to slow down `VACUUM` to reduce its impact on other users, though this increases the total duration of `VACUUM`.
- Improve B-tree index performance for duplicate keys (Dmitry Tkach, Tom)
 

This improves the way indexes are scanned when many duplicate values exist in the index.
- Use dynamically-generated table size estimates while planning (Tom)

Formerly the planner estimated table sizes using the values seen by the last `VACUUM` or `ANALYZE`, both as to physical table size (number of pages) and number of rows. Now, the current physical table size is obtained from the kernel, and the number of rows is estimated by multiplying the table size by the row density (rows per page) seen by the last `VACUUM` or `ANALYZE`. This should produce more reliable estimates in cases where the table size has changed significantly since the last housekeeping command.

- Improved index usage with `OR` clauses (Tom)

This allows the optimizer to use indexes in statements with many `OR` clauses that would not have been indexed in the past. It can also use multi-column indexes where the first column is specified and the second column is part of an `OR` clause.

- Improve matching of partial index clauses (Tom)

The server is now smarter about using partial indexes in queries involving complex `WHERE` clauses.

- Improve performance of the GEQO optimizer (Tom)

The GEQO optimizer is used to plan queries involving many tables (by default, twelve or more). This release speeds up the way queries are analyzed to decrease time spent in optimization.

- Miscellaneous optimizer improvements

There is not room here to list all the minor improvements made, but numerous special cases work better than in prior releases.

- Improve lookup speed for C functions (Tom)

This release uses a hash table to lookup information for dynamically loaded C functions. This improves their speed so they perform nearly as quickly as functions that are built into the server executable.

- Add type-specific `ANALYZE` statistics capability (Mark Cave-Ayland)

This feature allows more flexibility in generating statistics for non-standard data types.

- `ANALYZE` now collects statistics for expression indexes (Tom)

Expression indexes (also called functional indexes) allow users to index not just columns but the results of expressions and function calls. With this release, the optimizer can gather and use statistics about the contents of expression indexes. This will greatly improve the quality of planning for queries in which an expression index is relevant.

- New two-stage sampling method for `ANALYZE` (Manfred Koizar)

This gives better statistics when the density of valid rows is very different in different regions of a table.

- Speed up `TRUNCATE` (Tom)

This buys back some of the performance loss observed in 7.4, while still keeping `TRUNCATE` transaction-safe.

#### E.1.4.2. Server Changes

- Add WAL file archiving and point-in-time recovery (Simon Riggs)

- Add tablespaces so admins can control disk layout (Gavin)

- Add a built-in log rotation program (Andreas Pflug)

It is now possible to log server messages conveniently without relying on either `syslog` or an external log rotation program.

- Add new read-only server configuration parameters to show server compile-time settings: `block_size`, `integer_datetimes`, `max_function_args`, `max_identifier_length`, `max_index_keys` (Joe)

- Make quoting of `sameuser`, `samegroup`, and `all` remove special meaning of these terms in `pg_hba.conf` (Andrew)

- Use clearer IPv6 name `:::1/128` for `localhost` in default `pg_hba.conf` (Andrew)

- Use CIDR format in `pg_hba.conf` examples (Andrew)

- Rename server configuration parameters `SortMem` and `VacuumMem` to `work_mem` and `maintenance_work_mem` (Old names still supported) (Tom)

This change was made to clarify that bulk operations such as index and foreign key creation use `maintenance_work_mem`, while `work_mem` is for workspaces used during query execution.

- Allow logging of session disconnections using server configuration `log_disconnections` (Andrew)
- Add new server configuration parameter `log_line_prefix` to allow control of information emitted in each log line (Andrew)

Available information includes user name, database name, remote IP address, and session start time.

- Remove server configuration parameters `log_pid`, `log_timestamp`, `log_source_port`; functionality superseded by `log_line_prefix` (Andrew)
- Replace the `virtual_host` and `tcpip_socket` parameters with a unified `listen_addresses` parameter (Andrew, Tom)

`virtual_host` could only specify a single IP address to listen on. `listen_addresses` allows multiple addresses to be specified.

- Listen on localhost by default, which eliminates the need for the `-i` postmaster switch in many scenarios (Andrew)  
Listening on localhost (127.0.0.1) opens no new security holes but allows configurations like Windows and JDBC, which do not support local sockets, to work without special adjustments.
- Remove `syslog` server configuration parameter, and add more logical `log_destination` variable to control log output location (Magnus)
- Change server configuration parameter `log_statement` to take values `all`, `mod`, `ddl`, or `none` to select which queries are logged (Bruce)

This allows administrators to log only data definition changes or only data modification statements.

- Some logging-related configuration parameters could formerly be adjusted by ordinary users, but only in the “more verbose” direction. They are now treated more strictly: only superusers can set them. However, a superuser may use `ALTER USER` to provide per-user settings of these values for non-superusers. Also, it is now possible for superusers to set values of superuser-only configuration parameters via `PGOPTIONS`.
- Allow configuration files to be placed outside the data directory (mlw)

By default, configuration files are kept in the cluster's top directory. With this addition, configuration files can be placed outside the data directory, easing administration.

- Plan prepared queries only when first executed so constants can be used for statistics (Oliver Jowett)

Prepared statements plan queries once and execute them many times. While prepared queries avoid the overhead of re-planning on each use, the quality of the plan suffers from not knowing the exact parameters to be used in the query. In this release, planning of unnamed prepared statements is delayed until the first execution, and the actual parameter values of that execution are used as optimization hints. This allows use of out-of-line parameter passing without incurring a performance penalty.

- Allow `DECLARE CURSOR` to take parameters (Oliver Jowett)

It is now useful to issue `DECLARE CURSOR` in a `Parse` message with parameters. The parameter values sent at `Bind` time will be substituted into the execution of the cursor's query.

- Fix hash joins and aggregates of `inet` and `cidr` data types (Tom)

Release 7.4 handled hashing of mixed `inet` and `cidr` values incorrectly. (This bug did not exist in prior releases because they wouldn't try to hash either data type.)

- Make `log_duration` print only when `log_statement` prints the query (Ed L.)

### E.1.4.3. Query Changes

- Add savepoints (nested transactions) (Alvaro)
- Unsupported isolation levels are now accepted and promoted to the nearest supported level (Peter)

The SQL specification states that if a database doesn't support a specific isolation level, it should use the next more restrictive level. This change complies with that recommendation.

- Allow `BEGIN WORK` to specify transaction isolation levels like `START TRANSACTION` does (Bruce)
- Fix table permission checking for cases in which rules generate a query type different from the originally submitted query (Tom)
- Implement dollar quoting to simplify single-quote usage (Andrew, Tom, David Fetter)
 

In previous releases, because single quotes had to be used to quote a function's body, the use of single quotes inside the function text required use of two single quotes or other error-prone notations. With this release we add the ability to use "dollar quoting" to quote a block of text. The ability to use different quoting delimiters at different nesting levels greatly simplifies the task of quoting correctly, especially in complex functions. Dollar quoting can be used anywhere quoted text is needed.
- Make `CASE val WHEN compval1 THEN ... evaluate val` only once (Tom)
 

`CASE` no longer evaluates the tested expression multiple times. This has benefits when the expression is complex or is volatile.
- Test `HAVING` before computing target list of an aggregate query (Tom)
 

Fixes improper failure of cases such as `SELECT SUM(win)/SUM(lose) ... GROUP BY ... HAVING SUM(lose) > 0`. This should work but formerly could fail with divide-by-zero.
- Replace `max_expr_depth` parameter with `max_stack_depth` parameter, measured in kilobytes of stack size (Tom)
 

This gives us a fairly bulletproof defense against crashing due to runaway recursive functions. Instead of measuring the depth of expression nesting, we now directly measure the size of the execution stack.
- Allow arbitrary row expressions (Tom)
 

This release allows SQL expressions to contain arbitrary composite types, that is, row values. It also allows functions to more easily take rows as arguments and return row values.
- Allow `LIKE/ILIKE` to be used as the operator in row and subselect comparisons (Fabien Coelho)
- Avoid locale-specific case conversion of basic ASCII letters in identifiers and keywords (Tom)
 

This solves the “Turkish problem” with mangling of words containing `ı` and `i`. Folding of characters outside the 7-bit-ASCII set is still locale-aware.
- Improve syntax error reporting (Fabien, Tom)
 

Syntax error reports are more useful than before.
- Change `EXECUTE` to return a completion tag matching the executed statement (Kris Jurka)
 

Previous releases return an `EXECUTE` tag for any `EXECUTE` call. In this release, the tag returned will reflect the command executed.
- Avoid emitting `NATURAL CROSS JOIN` in rule listings (Tom)
 

Such a clause makes no logical sense, but in some cases the rule decompiler formerly produced this syntax.

#### E.1.4.4. Object Manipulation Changes

- Add `COMMENT ON` for casts, conversions, languages, operator classes, and large objects (Christopher)
- Add new server configuration parameter `default_with_oids` to control whether tables are created with `OIDs` by default (Neil)
 

This allows administrators to control whether `CREATE TABLE` commands create tables with or without `OID` columns by default. (Note: the current factory default setting for `default_with_oids` is `TRUE`, but the default will become `FALSE` in future releases.)
- Add `WITH / WITHOUT OIDS` clause to `CREATE TABLE AS` (Neil)
- Allow `ALTER TABLE DROP COLUMN` to drop an `OID` column (`ALTER TABLE SET WITHOUT OIDS` still works) (Tom)
- Allow composite types as table columns (Tom)
- Allow `ALTER ... ADD COLUMN` with defaults and `NOT NULL` constraints; works per SQL spec (Rod)

It is now possible for `ADD COLUMN` to create a column that is not initially filled with NULLs, but with a specified default value.

- Add `ALTER COLUMN TYPE` to change column's type (Rod)

It is now possible to alter a column's data type without dropping and re-adding the column.

- Allow multiple `ALTER` actions in a single `ALTER TABLE` command (Rod)

This is particularly useful for `ALTER` commands that rewrite the table (which include `ALTER COLUMN TYPE` and `ADD COLUMN` with a default). By grouping `ALTER` commands together, the table need be rewritten only once.

- Allow `ALTER TABLE` to add `SERIAL` columns (Tom)

This falls out from the new capability of specifying defaults for new columns.

- Allow changing the owners of aggregates, conversions, databases, functions, operators, operator classes, schemas, types, and tablespaces (Christopher, Euler Taveira de Oliveira)

Previously this required modifying the system tables directly.

- Allow temporary object creation to be limited to `SECURITY DEFINER` functions (Sean Chittenden)

- Add `ALTER TABLE ... SET WITHOUT CLUSTER` (Christopher)

Prior to this release, there was no way to clear an auto-cluster specification except to modify the system tables.

- Constraint/Index/SERIAL names are now *table\_column\_type* with numbers appended to guarantee uniqueness within the schema (Tom)

The SQL specification states that such names should be unique within a schema.

- Add `pg_get_serial_sequence()` to return a `SERIAL` column's sequence name (Christopher)

This allows automated scripts to reliably find the `SERIAL` sequence name.

- Warn when primary/foreign key data type mismatch requires costly lookup
- New `ALTER INDEX` command to allow moving of indexes between tablespaces (Gavin)
- Make `ALTER TABLE OWNER` change dependent sequence ownership too (Alvaro)

#### E.1.4.5. Utility Command Changes

- Allow `CREATE SCHEMA` to create triggers, indexes, and sequences (Neil)
- Add `ALSO` keyword to `CREATE RULE` (Fabien Coelho)

This allows `ALSO` to be added to rule creation to contrast it with `INSTEAD` rules.

- Add `NOWAIT` option to `LOCK` (Tatsuo)

This allows the `LOCK` command to fail if it would have to wait for the requested lock.

- Allow `COPY` to read and write comma-separated-value (CSV) files (Andrew, Bruce)
- Generate error if the `COPY` delimiter and `NULL` string conflict (Bruce)
- `GRANT/REVOKE` behavior follows the SQL spec more closely
- Avoid locking conflict between `CREATE INDEX` and `CHECKPOINT` (Tom)

In 7.3 and 7.4, a long-running B-tree index build could block concurrent `CHECKPOINTS` from completing, thereby causing WAL bloat because the WAL log could not be recycled.

- Database-wide `ANALYZE` does not hold locks across tables (Tom)

This reduces the potential for deadlocks against other backends that want exclusive locks on tables. To get the benefit of this change, do not execute database-wide `ANALYZE` inside a transaction block (`BEGIN` block); it must be able to commit and start a new transaction for each table.

- `REINDEX` does not exclusively lock the index's parent table anymore

The index itself is still exclusively locked, but readers of the table can continue if they are not using the particular index being rebuilt.

- Erase MD5 user passwords when a user is renamed (Bruce)

PostgreSQL uses the user name as salt when encrypting passwords via MD5. When a user's name is changed, the salt will no longer match the stored MD5 password, so the stored password becomes useless. In this release a notice is generated and the password is cleared. A new password must then be assigned if the user is to be able to log in with a password.

- New `pg_ctl kill` option for Windows (Andrew)

Windows does not have a `kill` command to send signals to backends so this capability was added to `pg_ctl`.

- Information schema improvements
- Add `--pwfile` option to `initdb` so the initial password can be set by GUI tools (Magnus)
- Detect locale/encoding mismatch in `initdb` (Peter)
- Add `register` command to `pg_ctl` to register Windows operating system service (Dave Page)

#### E.1.4.6. Data Type and Function Changes

- More complete support for composite types (row types) (Tom)

Composite values can be used in many places where only scalar values worked before.

- Reject non-rectangular array values as erroneous (Joe)

Formerly, `array_in` would silently build a surprising result.

- Overflow in integer arithmetic operations is now detected (Tom)

- The arithmetic operators associated with the single-byte `"char"` data type have been removed.

Formerly, the parser would select these operators in many situations where an “unable to select an operator” error would be more appropriate, such as `null * null`. If you actually want to do arithmetic on a `"char"` column, you can cast it to integer explicitly.

- Syntax checking of array input values considerably tightened up (Joe)

Junk that was previously allowed in odd places with odd results now causes an `ERROR`, for example, non-whitespace after the closing right brace.

- Empty-string array element values must now be written as `" "`, rather than writing nothing (Joe)

Formerly, both ways of writing an empty-string element value were allowed, but now a quoted empty string is required. The case where nothing at all appears will probably be considered to be a `NULL` element value in some future release.

- Array element trailing whitespace is now ignored (Joe)

Formerly leading whitespace was ignored, but trailing whitespace between an element value and the delimiter or right brace was significant. Now trailing whitespace is also ignored.

- Emit array values with explicit array bounds when lower bound is not one (Joe)

- Accept `YYYY-monthname-DD` as a date string (Tom)

- Make `netmask` and `hostmask` functions return maximum-length mask length (Tom)

- Change factorial function to return `numeric` (Gavin)

Returning `numeric` allows the factorial function to work for a wider range of input values.

- `to_char/to_date()` date conversion improvements (Kurt Roeckx, Fabien Coelho)

- Make `length()` disregard trailing spaces in `CHAR(n)` (Gavin)

This change was made to improve consistency: trailing spaces are semantically insignificant in `CHAR(n)` data, so they should not be counted by `length()`.

- Warn about empty string being passed to `OID/float4/float8` data types (Neil)

8.1 will throw an error instead.

- Allow leading or trailing whitespace in `int2/int4/int8/float4/float8` input routines (Neil)

- Better support for IEEE Infinity and NaN values in `float4/float8` (Neil)

These should now work on all platforms that support IEEE-compliant floating point arithmetic.

- Add `week` option to `date_trunc()` (Robert Creager)
- Fix `to_char` for 1 BC (previously it returned 1 AD) (Bruce)
- Fix `date_part(year)` for BC dates (previously it returned one less than the correct year) (Bruce)
- Fix `date_part()` to return the proper millennium and century (Fabien Coelho)

In previous versions, the century and millennium results had a wrong number and started in the wrong year, as compared to standard reckoning of such things.

- Add `ceiling()` as an alias for `ceil()`, and `power()` as an alias for `pow()` for standards compliance (Neil)
- Change `ln()`, `log()`, `power()`, and `sqrt()` to emit the correct `SQLSTATE` error codes for certain error conditions, as specified by SQL:2003 (Neil)
- Add `width_bucket()` function as defined by SQL:2003 (Neil)
- Add `generate_series()` functions to simplify working with numeric sets (Joe)
- Fix `upper/lower/initcap()` functions to work with multibyte encodings (Tom)
- Add boolean and bitwise integer AND/OR aggregates (Fabien Coelho)
- New session information functions to return network addresses for client and server (Sean Chittenden)
- Add function to determine the area of a closed path (Sean Chittenden)
- Add function to send cancel request to other backends (Magnus)
- Add `interval` plus `datetime` operators (Tom)

The reverse ordering, `datetime` plus `interval`, was already supported, but both are required by the SQL standard.

- Casting an integer to `BIT(N)` selects the rightmost N bits of the integer (Tom)

In prior releases, the leftmost N bits were selected, but this was deemed unhelpful, not to mention inconsistent with casting from bit to int.

- Require `CIDR` values to have all non-masked bits be zero (Kevin Brintnall)

#### E.1.4.7. Server-Side Language Changes

- In `READ COMMITTED` serialization mode, volatile functions now see the results of concurrent transactions committed up to the beginning of each statement within the function, rather than up to the beginning of the interactive command that called the function.
- Functions declared `STABLE` or `IMMUTABLE` always use the snapshot of the calling query, and therefore do not see the effects of actions taken after the calling query starts, whether in their own transaction or other transactions. Such a function must be read-only, too, meaning that it cannot use any SQL commands other than `SELECT`. There is a considerable performance gain from declaring a function `STABLE` or `IMMUTABLE` rather than `VOLATILE`.
- Non-deferred `AFTER` triggers are now fired immediately after completion of the triggering query, rather than upon finishing the current interactive command. This makes a difference when the triggering query occurred within a function: the trigger is invoked before the function proceeds to its next operation. For example, if a function inserts a new row into a table, any non-deferred foreign key checks occur before proceeding with the function.
- Allow function parameters to be declared with names (Dennis Bjorklund)

This allows better documentation of functions. Whether the names actually do anything depends on the specific function language being used.

- Allow PL/pgSQL parameter names to be referenced in the function (Dennis Bjorklund)

This basically creates an automatic alias for each named parameter.

- Do minimal syntax checking of PL/pgSQL functions at creation time (Tom)

This allows us to catch simple syntax errors sooner.

- More support for composite types (row and record variables) in PL/pgSQL  
For example, it now works to pass a rowtype variable to another function as a single variable.
- Default values for PL/pgSQL variables can now reference previously declared variables
- Improve parsing of PL/pgSQL FOR loops (Tom)  
Parsing is now driven by presence of ". ." rather than data type of FOR variable. This makes no difference for correct functions, but should result in more understandable error messages when a mistake is made.
- Major overhaul of PL/Perl server-side language (Command Prompt, Andrew Dunstan)
- In PL/Tcl, SPI commands are now run in subtransactions. If an error occurs, the subtransaction is cleaned up and the error is reported as an ordinary Tcl error, which can be trapped with `catch`. Formerly, it was not possible to catch such errors.
- Accept `ELSEIF` in PL/pgSQL (Neil)  
Previously PL/pgSQL only allowed `ELSIF`, but many people are accustomed to spelling this keyword `ELSEIF`.

#### E.1.4.8. **psql Changes**

- Improve psql information display about database objects (Christopher)
- Allow psql to display group membership in `\du` and `\dg` (Markus Bertheau)
- Prevent psql `\dn` from showing temporary schemas (Bruce)
- Allow psql to handle tilde user expansion for file names (Zach Irmen)
- Allow psql to display fancy prompts, including color, via readline (Reece Hart, Chet Ramey)
- Make psql `\copy` match `COPY` command syntax fully (Tom)
- Show the location of syntax errors (Fabien Coelho, Tom)
- Add `CLUSTER` information to psql `\d` display (Bruce)
- Change psql `\copy stdin/stdout` to read from command input/output (Bruce)
- Add `pstdin/pstdout` to read from psql's `stdin/stdout` (Mark Feit)
- Add global psql configuration file, `psqlrc.sample` (Bruce)  
This allows a central file where global psql startup commands can be stored.
- Have psql `\d+` indicate if the table has an `OID` column (Neil)
- On Windows, use binary mode in psql when reading files so control-Z is not seen as end-of-file
- Have `\dn+` show permissions and description for schemas (Dennis Bjorklund)
- Improve tab completion support (Stefan Kaltenbrunn, Greg Sabino Mullane)
- Allow boolean settings to be set using upper or lower case (Michael Paesold)

#### E.1.4.9. **pg\_dump Changes**

- Use dependency information to improve the reliability of `pg_dump` (Tom)  
This should solve the longstanding problems with related objects sometimes being dumped in the wrong order.
- Have `pg_dump` output objects in alphabetical order if possible (Tom)  
This should make it easier to identify changes between dump files.
- Allow `pg_restore` to ignore some SQL errors (Fabien Coelho)  
This makes `pg_restore`'s behavior similar to the results of feeding a `pg_dump` output script to psql. In most cases, ignoring errors and plowing ahead is the most useful thing to do. Also added was a `pg_restore` option to give the old behavior of exiting on an error.
- `pg_restore -l` display now includes objects' schema names
- New begin/end markers in `pg_dump` text output (Bruce)



- Add start/stop times for `pg_dump/pg_dumpall` in verbose mode (Bruce)
- Allow most `pg_dump` options in `pg_dumpall` (Christopher)
- Have `pg_dump` use `ALTER OWNER` rather than `SET SESSION AUTHORIZATION` by default (Christopher)

#### E.1.4.10. libpq Changes

- Make libpq's `SIGPIPE` handling thread-safe (Bruce)
- Add `PQmbdstrlen()` which returns the display length of a character (Tatsuo)
- Add thread locking to SSL and Kerberos connections (Manfred Spraul)
- Allow `PQoidValue()`, `PQcmdTuples()`, and `PQoidStatus()` to work on `EXECUTE` commands (Neil)
- Add `PQserverVersion()` to provide more convenient access to the server version number (Greg Sabino Mullane)
- Add `PQprepare/PQsendPrepared()` functions to support preparing statements without necessarily specifying the data types of their parameters (Abhijit Menon-Sen)
- Many ECPG improvements, including `SET DESCRIPTOR` (Michael)

#### E.1.4.11. Source Code Changes

- Allow the database server to run natively on Windows (Claudio, Magnus, Andrew)
- Shell script commands converted to C versions for Windows support (Andrew)
- Create an extension makefile framework (Fabien Coelho, Peter)

This simplifies the task of building extensions outside the original source tree.

- Support relocatable installations (Bruce)

Directory paths for installed files (such as the `/share` directory) are now computed relative to the actual location of the executables, so that an installation tree can be moved to another place without reconfiguring and rebuilding.

- Use `--with-docdir` to choose installation location of documentation; also allow `--infodir` (Peter)
- Add `--without-docdir` to prevent installation of documentation (Peter)
- Upgrade to DocBook V4.2 SGML (Peter)
- New PostgreSQL CVS tag (Marc)

This was done to make it easier for organizations to manage their own copies of the PostgreSQL CVS repository. File version stamps from the master repository will not get munged by checking into or out of a copied repository.

- Clarify locking code (Manfred Koizar)
- Buffer manager cleanup (Neil)
- Decouple platform tests from CPU spinlock code (Bruce, Tom)
- Add inlined test-and-set code on PA-RISC for gcc (ViSolve, Tom)
- Improve i386 spinlock code (Manfred Spraul)
- Clean up spinlock assembly code to avoid warnings from newer gcc releases (Tom)
- Remove JDBC from source tree; now a separate project
- Remove the libpqtel client interface; now a separate project
- More accurately estimate memory and file descriptor usage (Tom)
- Improvements to the Mac OS X startup scripts (Ray A.)
- New `fsync()` test program (Bruce)
- Major documentation improvements (Neil, Peter)
- Remove `pg_encoding`; not needed anymore
- Remove `pg_id`; not needed anymore
- Remove `initlocation`; not needed anymore

- Auto-detect thread flags (no more manual testing) (Bruce)
- Use Olson's public domain timezone library (Magnus)
- With threading enabled, use thread flags on Unixware for backend executables too (Bruce)  
Unixware can not mix threaded and non-threaded object files in the same executable, so everything must be compiled as threaded.
- psql now uses a flex-generated lexical analyzer to process command strings
- Reimplement the linked list data structure used throughout the backend (Neil)  
This improves performance by allowing list append and length operations to be more efficient.
- Allow dynamically loaded modules to create their own server configuration parameters (Thomas Hallgren)
- New Brazilian version of FAQ (Euler Taveira de Oliveira)
- Add French FAQ (Guillaume Lelarge)
- New pgevent for Windows logging
- Make libpq and ECPG build as proper shared libraries on OS X (Tom)

#### E.1.4.12. Contrib Changes

- Overhaul of contrib/dblink (Joe)
- contrib/dbmirror improvements (Steven Singer)
- New contrib/xml2 (John Gray, Torchbox)
- Updated contrib/mysql
- New version of contrib/btree\_gist (Teodor)
- New contrib/trgm, trigram matching for PostgreSQL (Teodor)
- Many contrib/tsearch2 improvements (Teodor)
- Add double metaphone to contrib/fuzzystrmatch (Andrew)
- Allow contrib/pg\_autovacuum to run as a Windows service (Dave Page)
- Add functions to contrib/dbsize (Andreas Pflug)
- Removed contrib/pg\_logger: obsoleted by integrated logging subprocess
- Removed contrib/rserve: obsoleted by various separate projects

## E.2. Release 7.4.6

**Release date:** 2004-10-22

This release contains a variety of fixes from 7.4.5.

### E.2.1. Migration to version 7.4.6

A dump/restore is not required for those running 7.4.X.

### E.2.2. Modificações

- Repair possible failure to update hint bits on disk  
Under rare circumstances this oversight could lead to “could not access transaction status” failures, which qualifies it as a potential-data-loss bug.
- Ensure that hashed outer join does not miss tuples  
Very large left joins using a hash join plan could fail to output unmatched left-side rows given just the right data distribution.
- Disallow running pg\_ctl as root

This is to guard against any possible security issues.

- Avoid using temp files in `/tmp` in `make_oidjoins_check`

This has been reported as a security issue, though it's hardly worthy of concern since there is no reason for non-developers to use this script anyway.

- Prevent forced backend shutdown from re-emitting prior command result

In rare cases, a client might think that its last command had succeeded when it really had been aborted by forced database shutdown.

- Repair bug in `pg_stat_get_backend_idset`

This could lead to misbehavior in some of the system-statistics views.

- Fix small memory leak in postmaster
- Fix “expected both swapped tables to have TOAST tables” bug

This could arise in cases such as `CLUSTER` after `ALTER TABLE DROP COLUMN`.

- Prevent `pg_ctl restart` from adding `-D` multiple times
- Fix problem with NULL values in GiST indexes
- `::` is no longer interpreted as a variable in an ECPG prepare statement

## E.3. Release 7.4.5

**Release date:** 2004-08-18

This release contains one serious bug fix over 7.4.4.

### E.3.1. Migration to version 7.4.5

A dump/restore is not required for those running 7.4.X.

### E.3.2. Modificações

- Repair possible crash during concurrent B-tree index insertions

This patch fixes a rare case in which concurrent insertions into a B-tree index could result in a server panic. No permanent damage would result, but it's still worth a re-release. The bug does not exist in pre-7.4 releases.

## E.4. Release 7.4.4

**Release date:** 2004-08-16

This release contains a variety of fixes from 7.4.3.

### E.4.1. Migration to version 7.4.4

A dump/restore is not required for those running 7.4.X.

### E.4.2. Modificações

- Prevent possible loss of committed transactions during crash

Due to insufficient interlocking between transaction commit and checkpointing, it was possible for transactions committed just before the most recent checkpoint to be lost, in whole or in part, following a database crash and restart. This is a serious bug that has existed since PostgreSQL 7.1.

- Check HAVING restriction before evaluating result list of an aggregate plan
- Avoid crash when session's current user ID is deleted
- Fix hashed crosstab for zero-rows case (Joe)

- Force cache update after renaming a column in a foreign key
- Pretty-print UNION queries correctly
- Make psql handle `\r\n` newlines properly in COPY IN
- `pg_dump` handled ACLs with grant options incorrectly
- Fix thread support for OS X and Solaris
- Updated JDBC driver (build 215) with various fixes
- ECPG fixes
- Translation updates (various contributors)

## E.5. Release 7.4.3

**Release date:** 2004-06-14

This release contains a variety of fixes from 7.4.2.

### E.5.1. Migration to version 7.4.3

A dump/restore is not required for those running 7.4.X.

### E.5.2. Modificações

- Fix temporary memory leak when using non-hashed aggregates (Tom)
- ECPG fixes, including some for Informix compatibility (Michael)
- Fixes for compiling with thread-safety, particularly Solaris (Bruce)
- Fix error in COPY IN termination when using the old network protocol (ljb)
- Several important fixes in `pg_autovacuum`, including fixes for large tables, unsigned oids, stability, temp tables, and debug mode (Matthew T. O'Connor)
- Fix problem with reading tar-format dumps on NetBSD and BSD/OS (Bruce)
- Several JDBC fixes
- Fix ALTER SEQUENCE RESTART where `last_value` equals the restart value (Tom)
- Repair failure to recalculate nested sub-selects (Tom)
- Fix problems with non-constant expressions in LIMIT/OFFSET
- Support FULL JOIN with no join clause, such as `X FULL JOIN Y ON TRUE` (Tom)
- Fix another zero-column table bug (Tom)
- Improve handling of non-qualified identifiers in GROUP BY clauses in sub-selects (Tom)  
Select-list aliases within the sub-select will now take precedence over names from outer query levels.
- Do not generate “NATURAL CROSS JOIN” when decompiling rules (Tom)
- Add checks for invalid field length in binary COPY (Tom)  
This fixes a difficult-to-exploit security hole.
- Avoid locking conflict between `ANALYZE` and `LISTEN/NOTIFY`
- Numerous translation updates (various contributors)

## E.6. Release 7.4.2

**Release date:** 2004-03-08

This release contains a variety of fixes from 7.4.1.

## E.6.1. Migration to version 7.4.2

A dump/restore is not required for those running 7.4.X. However, it may be advisable as the easiest method of incorporating fixes for two errors that have been found in the initial contents of 7.4.X system catalogs. A dump/initdb/reload sequence using 7.4.2's initdb will automatically correct these problems.

The more severe of the two errors is that data type `anyarray` has the wrong alignment label; this is a problem because the `pg_statistic` system catalog uses `anyarray` columns. The mislabeling can cause planner misestimations and even crashes when planning queries that involve `WHERE` clauses on double-aligned columns (such as `float8` and `timestamp`). It is strongly recommended that all installations repair this error, either by `initdb` or by following the manual repair procedure given below.

The lesser error is that the system view `pg_settings` ought to be marked as having public update access, to allow `UPDATE pg_settings` to be used as a substitute for `SET`. This can also be fixed either by `initdb` or manually, but it is not necessary to fix unless you want to use `UPDATE pg_settings`.

If you wish not to do an `initdb`, the following procedure will work for fixing `pg_statistic`. As the database superuser, do:

```
-- clear out old data in pg_statistic:
DELETE FROM pg_statistic;
VACUUM pg_statistic;
-- this should update 1 row:
UPDATE pg_type SET typalign = 'd' WHERE oid = 2277;
-- this should update 6 rows:
UPDATE pg_attribute SET attalign = 'd' WHERE atttypid = 2277;
--
-- At this point you MUST start a fresh backend to avoid a crash!
--
-- repopulate pg_statistic:
ANALYZE;
```

This can be done in a live database, but beware that all backends running in the altered database must be restarted before it is safe to repopulate `pg_statistic`.

To repair the `pg_settings` error, simply do:

```
GRANT SELECT, UPDATE ON pg_settings TO PUBLIC;
```

The above procedures must be carried out in *each* database of an installation, including `template1`, and ideally including `template0` as well. If you do not fix the template databases then any subsequently created databases will contain the same errors. `template1` can be fixed in the same way as any other database, but fixing `template0` requires additional steps. First, from any database issue

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

Next connect to `template0` and perform the above repair procedures. Finally, do

```
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

## E.6.2. Modificações

Release 7.4.2 incorporates all the fixes included in release 7.3.6, plus the following fixes:

- Fix `pg_statistics` alignment bug that could crash optimizer

See above for details about this problem.

- Allow non-super users to update `pg_settings`
- Fix several optimizer bugs, most of which led to “variable not found in subplan target lists” errors
- Avoid out-of-memory failure during startup of large multiple index scan
- Fix multibyte problem that could lead to “out of memory” error during `COPY IN`
- Fix problems with `SELECT INTO / CREATE TABLE AS` from tables without OIDs
- Fix problems with `alter_table` regression test during parallel testing
- Fix problems with hitting open file limit, especially on OS X (Tom)
- Partial fix for Turkish-locale issues  
initdb will succeed now in Turkish locale, but there are still some inconveniences associated with the `i / İ` problem.
- Make `pg_dump` set client encoding on restore
- Other minor `pg_dump` fixes
- Allow `ecpg` to again use C keywords as column names (Michael)
- Added `ecpg WHENEVER NOT_FOUND` to `SELECT/INSERT/UPDATE/DELETE` (Michael)
- Fix `ecpg` crash for queries calling set-returning functions (Michael)
- Various other `ecpg` fixes (Michael)
- Fixes for Borland compiler
- Thread build improvements (Bruce)
- Various other build fixes
- Various JDBC fixes

## E.7. Release 7.4.1

**Release date:** 2003-12-22

This release contains a variety of fixes from 7.4.

### E.7.1. Migration to version 7.4.1

A dump/restore is *not* required for those running 7.4.

If you want to install the fixes in the information schema you need to reload it into the database. This is either accomplished by initializing a new cluster by running `initdb`, or by running the following sequence of SQL commands in each database (ideally including `template1`) as a superuser in `psql`, after installing the new release:

```
DROP SCHEMA information_schema CASCADE;
\i /usr/local/pgsql/share/information_schema.sql
```

Substitute your installation path in the second command.

### E.7.2. Modificações

- Fixed bug in `CREATE SCHEMA` parsing in `ECPG` (Michael)
- Fix compile error when `--enable-thread-safety` and `--with-perl` are used together (Peter)
- Fix for subqueries that used hash joins (Tom)  
Certain subqueries that used hash joins would crash because of improperly shared structures.
- Fix free space map compaction bug (Tom)  
This fixes a bug where compaction of the free space map could lead to a database server shutdown.
- Fix for Borland compiler build of `libpq` (Bruce)
- Fix `netmask()` and `hostmask()` to return the maximum-length masklen (Tom)

Fix these functions to return values consistent with pre-7.4 releases.

- Several contrib/pg\_autovacuum fixes

Fixes include improper variable initialization, missing vacuum after TRUNCATE, and duration computation overflow for long vacuums.

- Allow compile of contrib/cube under Cygwin (Jason Tishler)
- Fix Solaris use of password file when no passwords are defined (Tom)

Fix crash on Solaris caused by use of any type of password authentication when no passwords were defined.

- JDBC fix for thread problems, other fixes
- Fix for bytea index lookups (Joe)
- Fix information schema for bit data types (Peter)
- Force zero\_damaged\_pages to be on during recovery from WAL
- Prevent some obscure cases of “variable not in subplan target lists”
- Make PQescapeBytea and byteaout consistent with each other (Joe)
- Escape bytea output for bytes > 0x7e (Joe)

If different client encodings are used for bytea output and input, it is possible for bytea values to be corrupted by the differing encodings. This fix escapes all bytes that might be affected.

- Added missing SPI\_finish() calls to dblink's get\_tuple\_of\_interest() (Joe)
- New Czech FAQ
- Fix information schema view constraint\_column\_usage for foreign keys (Peter)
- ECPG fixes (Michael)
- Fix bug with multiple IN subqueries and joins in the subqueries (Tom)
- Allow COUNT( 'x' ) to work (Tom)
- Install ECPG include files for Informix compatibility into separate directory (Peter)

Some names of ECPG include files for Informix compatibility conflicted with operating system include files. By installing them in their own directory, name conflicts have been reduced.

- Fix SSL memory leak (Neil)

This release fixes a bug in 7.4 where SSL didn't free all memory it allocated.

- Prevent pg\_service.conf from using service name as default dbname (Bruce)
- Fix local ident authentication on FreeBSD (Tom)

## E.8. Release 7.4

**Release date:** 2003-11-17

### E.8.1. Visão geral

Major changes in this release:

IN / NOT IN subqueries are now much more efficient

In previous releases, IN/NOT IN subqueries were joined to the upper query by sequentially scanning the subquery looking for a match. The 7.4 code uses the same sophisticated techniques used by ordinary joins and so is much faster. An IN will now usually be as fast as or faster than an equivalent EXISTS subquery; this reverses the conventional wisdom that applied to previous releases.

#### Improved `GROUP BY` processing by using hash buckets

In previous releases, rows to be grouped had to be sorted first. The 7.4 code can do `GROUP BY` without sorting, by accumulating results into a hash table with one entry per group. It will still use the sort technique, however, if the hash table is estimated to be too large to fit in `sort_mem`.

#### New multikey hash join capability

In previous releases, hash joins could only occur on single keys. This release allows multicolumn hash joins.

#### Queries using the explicit `JOIN` syntax are now better optimized

Prior releases evaluated queries using the explicit `JOIN` syntax only in the order implied by the syntax. 7.4 allows full optimization of these queries, meaning the optimizer considers all possible join orderings and chooses the most efficient. Outer joins, however, must still follow the declared ordering.

#### Faster and more powerful regular expression code

The entire regular expression module has been replaced with a new version by Henry Spencer, originally written for Tcl. The code greatly improves performance and supports several flavors of regular expressions.

#### Function-inlining for simple SQL functions

Simple SQL functions can now be inlined by including their SQL in the main query. This improves performance by eliminating per-call overhead. That means simple SQL functions now behave like macros.

#### Full support for IPv6 connections and IPv6 address data types

Previous releases allowed only IPv4 connections, and the IP data types only supported IPv4 addresses. This release adds full IPv6 support in both of these areas.

#### Major improvements in SSL performance and reliability

Several people very familiar with the SSL API have overhauled our SSL code to improve SSL key negotiation and error recovery.

#### Make free space map efficiently reuse empty index pages, and other free space management improvements

In previous releases, B-tree index pages that were left empty because of deleted rows could only be reused by rows with index values similar to the rows originally indexed on that page. In 7.4, `VACUUM` records empty index pages and allows them to be reused for any future index rows.

#### SQL-standard information schema

The information schema provides a standardized and stable way to access information about the schema objects defined in a database.

#### Cursors conform more closely to the SQL standard

The commands `FETCH` and `MOVE` have been overhauled to conform more closely to the SQL standard.

#### Cursors can exist outside transactions

These cursors are also called holdable cursors.

#### New client-to-server protocol

The new protocol adds error codes, more status information, faster startup, better support for binary data transmission, parameter values separated from SQL commands, prepared statements available at the protocol level, and cleaner recovery from `COPY` failures. The older protocol is still supported by both server and clients.

#### libpq and ECPG applications are now fully thread-safe

While previous libpq releases already supported threads, this release improves thread safety by fixing some non-thread-safe code that was used during database connection startup. The `configure` option `--enable-thread-safety` must be used to enable this feature.

#### New version of full-text indexing

A new full-text indexing suite is available in `contrib/tsearch2`.



### New autovacuum tool

The new autovacuum tool in contrib/autovacuum monitors the database statistics tables for INSERT/UPDATE/DELETE activity and automatically vacuums tables when needed.

Array handling has been improved and moved into the server core

Many array limitations have been removed, and arrays behave more like fully-supported data types.

## E.8.2. Migration to version 7.4

A dump/restore using `pg_dump` is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

- The server-side autocommit setting was removed and reimplemented in client applications and languages. Server-side autocommit was causing too many problems with languages and applications that wanted to control their own autocommit behavior, so autocommit was removed from the server and added to individual client APIs as appropriate.
- Error message wording has changed substantially in this release. Significant effort was invested to make the messages more consistent and user-oriented. If your applications try to detect different error conditions by parsing the error message, you are strongly encouraged to use the new error code facility instead.
- Inner joins using the explicit `JOIN` syntax may behave differently because they are now better optimized.
- A number of server configuration parameters have been renamed for clarity, primarily those related to logging.
- `FETCH 0` or `MOVE 0` now does nothing. In prior releases, `FETCH 0` would fetch all remaining rows, and `MOVE 0` would move to the end of the cursor.
- `FETCH` and `MOVE` now return the actual number of rows fetched/moved, or zero if at the beginning/end of the cursor. Prior releases would return the row count passed to the command, not the number of rows actually fetched or moved.
- `COPY` now can process files that use carriage-return or carriage-return/line-feed end-of-line sequences. Literal carriage-returns and line-feeds are no longer accepted in data values; use `\r` and `\n` instead.
- Trailing spaces are now trimmed when converting from type `char(n)` to `varchar(n)` or `text`. This is what most people always expected to happen anyway.
- The data type `float(p)` now measures *p* in binary digits, not decimal digits. The new behavior follows the SQL standard.
- Ambiguous date values now must match the ordering specified by the `datestyle` setting. In prior releases, a date specification of `10/20/03` was interpreted as a date in October even if `datestyle` specified that the day should be first. 7.4 will throw an error if a date specification is invalid for the current setting of `datestyle`.
- The functions `oidrand`, `oidsrand`, and `userfntest` have been removed. These functions were determined to be no longer useful.
- String literals specifying time-varying date/time values, such as `'now'` or `'today'` will no longer work as expected in column default expressions; they now cause the time of the table creation to be the default, not the time of the insertion. Functions such as `now()`, `current_timestamp`, or `current_date` should be used instead.

In previous releases, there was special code so that strings such as `'now'` were interpreted at `INSERT` time and not at table creation time, but this work around didn't cover all cases. Release 7.4 now requires that defaults be defined properly using functions such as `now()` or `current_timestamp`. These will work in all situations.

- The dollar sign (\$) is no longer allowed in operator names. It can instead be a non-first character in identifiers. This was done to improve compatibility with other database systems, and to avoid syntax problems when parameter placeholders (`$n`) are written adjacent to operators.

## E.8.3. Modificações

Below you will find a detailed account of the changes between release 7.4 and the previous major release.

**E.8.3.1. Server Operation Changes**

- Allow IPv6 server connections (Nigel Kukard, Johan Jordaan, Bruce, Tom, Kurt Roeckx, Andrew Dunstan)
- Fix SSL to handle errors cleanly (Nathan Mueller)

In prior releases, certain SSL API error reports were not handled correctly. This release fixes those problems.

- SSL protocol security and performance improvements (Sean Chittenden)

SSL key renegotiation was happening too frequently, causing poor SSL performance. Also, initial key handling was improved.

- Print lock information when a deadlock is detected (Tom)

This allows easier debugging of deadlock situations.

- Update `/tmp` socket modification times regularly to avoid their removal (Tom)

This should help prevent `/tmp` directory cleaner administration scripts from removing server socket files.

- Enable PAM for Mac OS X (Aaron Hillegass)

- Make B-tree indexes fully WAL-safe (Tom)

In prior releases, under certain rare cases, a server crash could cause B-tree indexes to become corrupt. This release removes those last few rare cases.

- Allow B-tree index compaction and empty page reuse (Tom)
- Fix inconsistent index lookups during split of first root page (Tom)

In prior releases, when a single-page index split into two pages, there was a brief period when another database session could miss seeing an index entry. This release fixes that rare failure case.

- Improve free space map allocation logic (Tom)
- Preserve free space information between server restarts (Tom)

In prior releases, the free space map was not saved when the postmaster was stopped, so newly started servers had no free space information. This release saves the free space map, and reloads it when the server is restarted.

- Add start time to `pg_stat_activity` (Neil)
- New code to detect corrupt disk pages; erase with `zero_damaged_pages` (Tom)
- New client/server protocol: faster, no username length limit, allow clean exit from `COPY` (Tom)
- Add transaction status, table ID, column ID to client/server protocol (Tom)
- Add binary I/O to client/server protocol (Tom)
- Remove autocommit server setting; move to client applications (Tom)
- New error message wording, error codes, and three levels of error detail (Tom, Joe, Peter)

**E.8.3.2. Performance Improvements**

- Add hashing for `GROUP BY` aggregates (Tom)
- Make nested-loop joins be smarter about multicolumn indexes (Tom)
- Allow multikey hash joins (Tom)
- Improve constant folding (Tom)
- Add ability to inline simple SQL functions (Tom)
- Reduce memory usage for queries using complex functions (Tom)

In prior releases, functions returning allocated memory would not free it until the query completed. This release allows the freeing of function-allocated memory when the function call completes, reducing the total memory used by functions.

- Improve GEQO optimizer performance (Tom)

This release fixes several inefficiencies in the way the GEQO optimizer manages potential query paths.

- Allow `IN/NOT IN` to be handled via hash tables (Tom)
- Improve `NOT IN (subquery)` performance (Tom)
- Allow most `IN` subqueries to be processed as joins (Tom)
- Pattern matching operations can use indexes regardless of locale (Peter)
 

There is no way for non-ASCII locales to use the standard indexes for `LIKE` comparisons. This release adds a way to create a special index for `LIKE`.
- Allow the postmaster to preload libraries using `preload_libraries` (Joe)
 

For shared libraries that require a long time to load, this option is available so the library can be preloaded in the postmaster and inherited by all database sessions.
- Improve optimizer cost computations, particularly for subqueries (Tom)
- Avoid sort when subquery `ORDER BY` matches upper query (Tom)
- Deduce that `WHERE a.x = b.y AND b.y = 42` also means `a.x = 42` (Tom)
- Allow hash/merge joins on complex joins (Tom)
- Allow hash joins for more data types (Tom)
- Allow join optimization of explicit inner joins, disable with `join_collapse_limit` (Tom)
- Add parameter `from_collapse_limit` to control conversion of subqueries to joins (Tom)
- Use faster and more powerful regular expression code from Tcl (Henry Spencer, Tom)
- Use bit-mapped relation sets in the optimizer (Tom)
- Improve connection startup time (Tom)
 

The new client/server protocol requires fewer network packets to start a database session.
- Improve trigger/constraint performance (Stephan)
- Improve speed of `col IN (const, const, const, ...)` (Tom)
- Fix hash indexes which were broken in rare cases (Tom)
- Improve hash index concurrency and speed (Tom)
 

Prior releases suffered from poor hash index performance, particularly for high concurrency situations. This release fixes that, and the development group is interested in reports comparing B-tree and hash index performance.
- Align shared buffers on 32-byte boundary for copy speed improvement (Manfred Spraul)
 

Certain CPU's perform faster data copies when addresses are 32-byte aligned.
- Data type `numeric` reimplemented for better performance (Tom)
 

`numeric` used to be stored in base 100. The new code uses base 10000, for significantly better performance.

### E.8.3.3. Server Configuration Changes

- Rename server parameter `server_min_messages` to `log_min_messages` (Bruce)
 

This was done so most parameters that control the server logs begin with `log_`.
- Rename `show_*_stats` to `log_*_stats` (Bruce)
- Rename `show_source_port` to `log_source_port` (Bruce)
- Rename `hostname_lookup` to `log_hostname` (Bruce)
- Add `checkpoint_warning` to warn of excessive checkpointing (Bruce)
 

In prior releases, it was difficult to determine if checkpoint was happening too frequently. This feature adds a warning to the server logs when excessive checkpointing happens.
- New read-only server parameters for localization (Tom)
- Change debug server log messages to output as `DEBUG` rather than `LOG` (Bruce)

- Prevent server log variables from being turned off by non-superusers (Bruce)  
This is a security feature so non-superusers cannot disable logging that was enabled by the administrator.
- `log_min_messages/client_min_messages` now controls `debug_*` output (Bruce)  
This centralizes client debug information so all debug output can be sent to either the client or server logs.
- Add Mac OS X Rendezvous server support (Chris Campbell)  
This allows Mac OS X hosts to query the network for available PostgreSQL servers.
- Add ability to print only slow statements using `log_min_duration_statement` (Christopher)  
This is an often requested debugging feature that allows administrators to see only slow queries in their server logs.
- Allow `pg_hba.conf` to accept netmasks in CIDR format (Andrew Dunstan)  
This allows administrators to merge the host IP address and netmask fields into a single CIDR field in `pg_hba.conf`.
- New read-only parameter `is_superuser` (Tom)
- New parameter `log_error_verbosity` to control error detail (Tom)  
This works with the new error reporting feature to supply additional error information like hints, file names and line numbers.
- `postgres --describe-config` now dumps server config variables (Aizaz Ahmed, Peter)  
This option is useful for administration tools that need to know the configuration variable names and their minimums, maximums, defaults, and descriptions.
- Add new columns in `pg_settings`: `context`, `type`, `source`, `min_val`, `max_val` (Joe)
- Make default `shared_buffers` 1000 and `max_connections` 100, if possible (Tom)  
Prior versions defaulted to 64 shared buffers so PostgreSQL would start on even very old systems. This release tests the amount of shared memory allowed by the platform and selects more reasonable default values if possible. Of course, users are still encouraged to evaluate their resource load and size `shared_buffers` accordingly.
- New `pg_hba.conf` record type `hostnossl` to prevent SSL connections (Jon Jensen)  
In prior releases, there was no way to prevent SSL connections if both the client and server supported SSL. This option allows that capability.
- Remove parameter `geqo_random_seed` (Tom)
- Add server parameter `regex_flavor` to control regular expression processing (Tom)
- Make `pg_ctl` better handle nonstandard ports (Greg)

#### E.8.3.4. Query Changes

- New SQL-standard information schema (Peter)
- Add read-only transactions (Peter)
- Print key name and value in foreign-key violation messages (Dmitry Tkach)
- Allow users to see their own queries in `pg_stat_activity` (Kevin Brown)  
In prior releases, only the superuser could see query strings using `pg_stat_activity`. Now ordinary users can see their own query strings.
- Fix aggregates in subqueries to match SQL standard (Tom)  
The SQL standard says that an aggregate function appearing within a nested subquery belongs to the outer query if its argument contains only outer-query variables. Prior PostgreSQL releases did not handle this fine point correctly.
- Add option to prevent auto-addition of tables referenced in query (Nigel J. Andrews)  
By default, tables mentioned in the query are automatically added to the `FROM` clause if they are not already there. This is compatible with historic POSTGRES behavior but is contrary to the SQL standard. This option allows selecting standard-compatible behavior.

- Allow `UPDATE ... SET col = DEFAULT` (Rod)  
This allows `UPDATE` to set a column to its declared default value.
- Allow expressions to be used in `LIMIT/OFFSET` (Tom)  
In prior releases, `LIMIT/OFFSET` could only use constants, not expressions.
- Implement `CREATE TABLE AS EXECUTE` (Neil, Peter)

### E.8.3.5. Object Manipulation Changes

- Make `CREATE SEQUENCE` grammar more conforming to SQL:2003 (Neil)
- Add statement-level triggers (Neil)  
While this allows a trigger to fire at the end of a statement, it does not allow the trigger to access all rows modified by the statement. This capability is planned for a future release.
- Add check constraints for domains (Rod)  
This greatly increases the usefulness of domains by allowing them to use check constraints.
- Add `ALTER DOMAIN` (Rod)  
This allows manipulation of existing domains.
- Fix several zero-column table bugs (Tom)  
PostgreSQL supports zero-column tables. This fixes various bugs that occur when using such tables.
- Have `ALTER TABLE ... ADD PRIMARY KEY` add not-null constraint (Rod)  
In prior releases, `ALTER TABLE ... ADD PRIMARY` would add a unique index, but not a not-null constraint. That is fixed in this release.
- Add `ALTER TABLE ... WITHOUT OIDS` (Rod)  
This allows control over whether new and updated rows will have an OID column. This is most useful for saving storage space.
- Add `ALTER SEQUENCE` to modify minimum, maximum, increment, cache, cycle values (Rod)
- Add `ALTER TABLE ... CLUSTER ON` (Alvaro Herrera)  
This command is used by `pg_dump` to record the cluster column for each table previously clustered. This information is used by database-wide cluster to cluster all previously clustered tables.
- Improve automatic type casting for domains (Rod, Tom)
- Allow dollar signs in identifiers, except as first character (Tom)
- Disallow dollar signs in operator names, so `x=$1` works (Tom)
- Allow copying table schema using `LIKE subtable`, also SQL:2003 feature `INCLUDING DEFAULTS` (Rod)
- Add `WITH GRANT OPTION` clause to `GRANT` (Peter)  
This enabled `GRANT` to give other users the ability to grant privileges on a object.

### E.8.3.6. Utility Command Changes

- Add `ON COMMIT` clause to `CREATE TABLE` for temporary tables (Gavin)  
This adds the ability for a table to be dropped or all rows deleted on transaction commit.
- Allow cursors outside transactions using `WITH HOLD` (Neil)  
In previous releases, cursors were removed at the end of the transaction that created them. Cursors can now be created with the `WITH HOLD` option, which allows them to continue to be accessed after the creating transaction has committed.
- `FETCH 0` and `MOVE 0` now do nothing (Bruce)  
In previous releases, `FETCH 0` fetched all remaining rows, and `MOVE 0` moved to the end of the cursor.

- Cause `FETCH` and `MOVE` to return the number of rows fetched/moved, or zero if at the beginning/end of cursor, per SQL standard (Bruce)

In prior releases, the row count returned by `FETCH` and `MOVE` did not accurately reflect the number of rows processed.

- Properly handle `SCROLL` with cursors, or report an error (Neil)

Allowing random access (both forward and backward scrolling) to some kinds of queries cannot be done without some additional work. If `SCROLL` is specified when the cursor is created, this additional work will be performed. Furthermore, if the cursor has been created with `NO SCROLL`, no random access is allowed.

- Implement SQL-compatible options `FIRST`, `LAST`, `ABSOLUTE n`, `RELATIVE n` for `FETCH` and `MOVE` (Tom)
- Allow `EXPLAIN ON DECLARE CURSOR` (Tom)
- Allow `CLUSTER` to use index marked as pre-clustered by default (Alvaro Herrera)
- Allow `CLUSTER` to cluster all tables (Alvaro Herrera)

This allows all previously clustered tables in a database to be reclustered with a single command.

- Prevent `CLUSTER` on partial indexes (Tom)
- Allow DOS and Mac line-endings in `COPY` files (Bruce)
- Disallow literal carriage return as a data value, backslash-carriage-return and `\r` are still allowed (Bruce)
- `COPY` changes (binary, `\.`) (Tom)
- Recover from `COPY` failure cleanly (Tom)
- Prevent possible memory leaks in `COPY` (Tom)
- Make `TRUNCATE` transaction-safe (Rod)

`TRUNCATE` can now be used inside a transaction. If the transaction aborts, the changes made by the `TRUNCATE` are automatically rolled back.

- Allow prepare/bind of utility commands like `FETCH` and `EXPLAIN` (Tom)
- Add `EXPLAIN EXECUTE` (Neil)
- Improve `VACUUM` performance on indexes by reducing WAL traffic (Tom)
- Functional indexes have been generalized into indexes on expressions (Tom)

In prior releases, functional indexes only supported a simple function applied to one or more column names. This release allows any type of scalar expression.

- Have `SHOW TRANSACTION ISOLATION` match input to `SET TRANSACTION ISOLATION` (Tom)
- Have `COMMENT ON DATABASE` on nonlocal database generate a warning, rather than an error (Rod)

Database comments are stored in database-local tables so comments on a database have to be stored in each database.

- Improve reliability of `LISTEN/NOTIFY` (Tom)
- Allow `REINDEX` to reliably reindex nonshared system catalog indexes (Tom)

This allows system tables to be reindexed without the requirement of a standalone session, which was necessary in previous releases. The only tables that now require a standalone session for reindexing are the global system tables `pg_database`, `pg_shadow`, and `pg_group`.

### E.8.3.7. Data Type and Function Changes

- New server parameter `extra_float_digits` to control precision display of floating-point numbers (Pedro Ferreira, Tom)

This controls output precision which was causing regression testing problems.

- Allow `+1300` as a numeric time-zone specifier, for `FJST` (Tom)
- Remove rarely used functions `oidrand`, `oidsrand`, and `userfntest` functions (Neil)
- Add `md5()` function to main server, already in `contrib/pgcrypto` (Joe)

An MD5 function was frequently requested. For more complex encryption capabilities, use `contrib/pgcrypto`.

- Increase date range of `timestamp` (John Cochran)
- Change `EXTRACT(EPOCH FROM timestamp)` so `timestamp` without time zone is assumed to be in local time, not GMT (Tom)
- Trap division by zero in case the operating system doesn't prevent it (Tom)
- Change the numeric data type internally to base 10000 (Tom)
- New `hostmask()` function (Greg Wickham)
- Fixes for `to_char()` and `to_timestamp()` (Karel)
- Allow functions that can take any argument data type and return any data type, using `anyelement` and `anyarray` (Joe)
 

This allows the creation of functions that can work with any data type.
- Arrays may now be specified as `ARRAY[1,2,3]`, `ARRAY[['a','b'],['c','d']]`, or `ARRAY[ARRAY[ARRAY[2]]]` (Joe)
- Allow proper comparisons for arrays, including `ORDER BY` and `DISTINCT` support (Joe)
- Allow indexes on array columns (Joe)
- Allow array concatenation with `||` (Joe)
- Allow `WHERE` qualification `expr op ANY/SOME/ALL (array_expr)` (Joe)
 

This allows arrays to behave like a list of values, for purposes like `SELECT * FROM tab WHERE col IN (array_val)`.
- New array functions `array_append`, `array_cat`, `array_lower`, `array_prepend`, `array_to_string`, `array_upper`, `string_to_array` (Joe)
- Allow user defined aggregates to use polymorphic functions (Joe)
- Allow assignments to empty arrays (Joe)
- Allow 60 in seconds fields of time, `timestamp`, and interval input values (Tom)
 

Sixty-second values are needed for leap seconds.
- Allow `cidr` data type to be cast to `text` (Tom)
- Disallow invalid time zone names in `SET TIMEZONE`
- Trim trailing spaces when `char` is cast to `varchar` or `text` (Tom)
- Make `float(p)` measure the precision  $p$  in binary digits, not decimal digits (Tom)
- Add IPv6 support to the `inet` and `cidr` data types (Michael Graff)
- Add `family()` function to report whether address is IPv4 or IPv6 (Michael Graff)
- Have `SHOW datestyle` generate output similar to that used by `SET datestyle` (Tom)
- Make `EXTRACT(TIMEZONE)` and `SET/SHOW TIME ZONE` follow the SQL convention for the sign of time zone offsets, i.e., positive is east from UTC (Tom)
- Fix `date_trunc('quarter', ...)` (Böjthe Zoltán)
 

Prior releases returned an incorrect value for this function call.
- Make `initcap()` more compatible with Oracle (Mike Nolan)
 

`initcap()` now uppercases a letter appearing after any non-alphanumeric character, rather than only after whitespace.
- Allow only `datestyle` field order for date values not in ISO-8601 format (Greg)
- Add new `datestyle` values `MDY`, `DMY`, and `YMD` to set input field order; honor `US` and `European` for backward compatibility (Tom)
- String literals like `'now'` or `'today'` will no longer work as a column default. Use functions such as `now()`, `current_timestamp` instead. (change required for prepared statements) (Tom)

- Treat NaN as larger than any other value in `min()`/`max()` (Tom)  
NaN was already sorted after ordinary numeric values for most purposes, but `min()` and `max()` didn't get this right.
- Prevent interval from suppressing `:00` seconds display
- New functions `pg_get_triggerdef(prettyprint)` and `pg_conversion_is_visible()` (Christopher)
- Allow time to be specified as `040506` or `0405` (Tom)
- Input date order must now be `YYYY-MM-DD` (with 4-digit year) or match `datestyle`
- Make `pg_get_constraintdef` support unique, primary-key, and check constraints (Christopher)

#### E.8.3.8. Server-Side Language Changes

- Prevent PL/pgSQL crash when `RETURN NEXT` is used on a zero-row record variable (Tom)
- Make PL/Python's `spi_execute` interface handle null values properly (Andrew Bosma)
- Allow PL/pgSQL to declare variables of composite types without `%ROWTYPE` (Tom)
- Fix PL/Python's `_quote()` function to handle big integers
- Make PL/Python an untrusted language, now called `plpythonu` (Kevin Jacobs, Tom)  
The Python language no longer supports a restricted execution environment, so the trusted version of PL/Python was removed. If this situation changes, a version of PL/Python that can be used by non-superusers will be readded.
- Allow polymorphic PL/pgSQL functions (Joe, Tom)
- Allow polymorphic SQL functions (Joe)
- Improved compiled function caching mechanism in PL/pgSQL with full support for polymorphism (Joe)
- Add new parameter `$0` in PL/pgSQL representing the function's actual return type (Joe)
- Allow PL/Tcl and PL/Python to use the same trigger on multiple tables (Tom)
- Fixed PL/Tcl's `spi_prepare` to accept fully qualified type names in the parameter type list (Jan)

#### E.8.3.9. psql Changes

- Add `\pset pager always` to always use pager (Greg)  
This forces the pager to be used even if the number of rows is less than the screen height. This is valuable for rows that wrap across several screen rows.
- Improve tab completion (Rod, Ross Reedstrom, Ian Barwick)
- Reorder `\?` help into groupings (Harald Armin Massa, Bruce)
- Add backslash commands for listing schemas, casts, and conversions (Christopher)
- `\encoding` now changes based on the server parameter `client_encoding` (Tom)  
In previous versions, `\encoding` was not aware of encoding changes made using `SET client_encoding`.
- Save editor buffer into readline history (Ross)  
When `\e` is used to edit a query, the result is saved in the readline history for retrieval using the up arrow.
- Improve `\d` display (Christopher)
- Enhance HTML mode to be more standards-conforming (Greg)
- New `\set AUTOCOMMIT off` capability (Tom)  
This takes the place of the removed server parameter `autocommit`.
- New `\set VERBOSITY` to control error detail (Tom)  
This controls the new error reporting details.
- New prompt escape sequence `%x` to show transaction status (Tom)
- Long options for psql are now available on all platforms



**E.8.3.10. pg\_dump Changes**

- Multiple `pg_dump` fixes, including tar format and large objects
- Allow `pg_dump` to dump specific schemas (Neil)
- Make `pg_dump` preserve column storage characteristics (Christopher)
 

This preserves `ALTER TABLE ... SET STORAGE` information.
- Make `pg_dump` preserve `CLUSTER` characteristics (Christopher)
- Have `pg_dumpall` use `GRANT/REVOKE` to dump database-level privileges (Tom)
- Allow `pg_dumpall` to support the options `-a`, `-s`, `-x` of `pg_dump` (Tom)
- Prevent `pg_dump` from lowercasing identifiers specified on the command line (Tom)
- `pg_dump` options `--use-set-session-authorization` and `--no-reconnect` now do nothing, all dumps use `SET SESSION AUTHORIZATION`

`pg_dump` no longer reconnects to switch users, but instead always uses `SET SESSION AUTHORIZATION`. This will reduce password prompting during restores.
- Long options for `pg_dump` are now available on all platforms
 

PostgreSQL now includes its own long-option processing routines.

**E.8.3.11. libpq Changes**

- Add function `PQfreemem` for freeing memory on Windows, suggested for `NOTIFY` (Bruce)
 

Windows requires that memory allocated in a library be freed by a function in the same library, hence `free()` doesn't work for freeing memory allocated by `libpq`. `PQfreemem` is the proper way to free `libpq` memory, especially on Windows, and is recommended for other platforms as well.
- Document service capability, and add sample file (Bruce)
 

This allows clients to look up connection information in a central file on the client machine.
- Make `PQsetdbLogin` have the same defaults as `PQconnectdb` (Tom)
- Allow `libpq` to cleanly fail when result sets are too large (Tom)
- Improve performance of function `PQunescapeBytea` (Ben Lamb)
- Allow thread-safe `libpq` with `configure` option `--enable-thread-safety` (Lee Kindness, Philip Yarra)
- Allow function `pqInternalNotice` to accept a format string and arguments instead of just a preformatted message (Tom, Sean Chittenden)
- Control SSL negotiation with `sslmode` values `disable`, `allow`, `prefer`, and `require` (Jon Jensen)
- Allow new error codes and levels of text (Tom)
- Allow access to the underlying table and column of a query result (Tom)
 

This is helpful for query-builder applications that want to know the underlying table and column names associated with a specific result set.
- Allow access to the current transaction status (Tom)
- Add ability to pass binary data directly to the server (Tom)
- Add function `PQexecPrepared` and `PQsendQueryPrepared` functions which perform `bind/execute` of previously prepared statements (Tom)

**E.8.3.12. JDBC Changes**

- Allow `setNull` on updateable result sets
- Allow `executeBatch` on a prepared statement (Barry)
- Support SSL connections (Barry)
- Handle schema names in result sets (Paul Sorenson)

- Add refcursor support (Nic Ferrier)

### E.8.3.13. Miscellaneous Interface Changes

- Prevent possible memory leak or core dump during libpqctl shutdown (Tom)
- Add Informix compatibility to ECPG (Michael)
 

This allows ECPG to process embedded C programs that were written using certain Informix extensions.
- Add type `decimal` to ECPG that is fixed length, for Informix (Michael)
- Allow thread-safe embedded SQL programs with `configure` option `--enable-thread-safety` (Lee Kindness, Bruce)
 

This allows multiple threads to access the database at the same time.
- Moved Python client PyGreSQL to <http://www.pygresql.org> (Marc)

### E.8.3.14. Source Code Changes

- Prevent need for separate platform geometry regression result files (Tom)
- Improved PPC locking primitive (Reinhard Max)
- New function `palloc0` to allocate and clear memory (Bruce)
- Fix locking code for s390x CPU (64-bit) (Tom)
- Allow OpenBSD to use local ident credentials (William Ahern)
- Make query plan trees read-only to executor (Tom)
- Add Darwin startup scripts (David Wheeler)
- Allow libpq to compile with Borland C++ compiler (Lester Godwin, Karl Waclawek)
- Use our own version of `getopt_long()` if needed (Peter)
- Convert administration scripts to C (Peter)
- Bison `>= 1.85` is now required to build the PostgreSQL grammar, if building from CVS
- Merge documentation into one book (Peter)
- Add Windows compatibility functions (Bruce)
- Allow client interfaces to compile under MinGW (Bruce)
- New `ereport()` function for error reporting (Tom)
- Support Intel compiler on Linux (Peter)
- Improve Linux startup scripts (Slawomir Sudnik, Darko Prenosil)
- Add support for AMD Opteron and Itanium (Jeffrey W. Baker, Bruce)
- Remove `--enable-recode` option from `configure`

This was no longer needed now that we have `CREATE CONVERSION`.
- Generate a compile error if spinlock code is not found (Bruce)
 

Platforms without spinlock code will now fail to compile, rather than silently using semaphores. This failure can be disabled with a new `configure` option.

### E.8.3.15. Contrib Changes

- Change dbmirror license to BSD
- Improve earthdistance (Bruno Wolff III)
- Portability improvements to pgcrypto (Marko Kreen)
- Prevent crash in xml (John Gray, Michael Richards)
- Update oracle

- Update mysql
- Update cube (Bruno Wolff III)
- Update earthdistance to use cube (Bruno Wolff III)
- Update btree\_gist (Oleg)
- New tsearch2 full-text search module (Oleg, Teodor)
- Add hash-based crosstab function to tablefuncs (Joe)
- Add serial column to order connectby( ) siblings in tablefuncs (Nabil Sayegh, Joe)
- Add named persistent connections to dblink (Shridhar Daithanka)
- New pg\_autovacuum allows automatic VACUUM (Matthew T. O'Connor)
- Make pgbench honor environment variables PGHOST, PGPORT, PGUSER (Tatsuo)
- Improve intarray (Teodor Sigaev)
- Improve pgstattuple (Rod)
- Fix bug in metaphone( ) in fuzzystmatch
- Improve adddepend (Rod)
- Update spi/timetravel (Böjthe Zoltán)
- Fix dbase -s option and improve non-ASCII handling (Thomas Behr, Márcio Smiderle)
- Remove array module because features now included by default (Joe)

## E.9. Release 7.3.8

**Release date:** 2004-10-22

This release contains a variety of fixes from 7.3.7.

### E.9.1. Migration to version 7.3.8

A dump/restore is not required for those running 7.3.X.

### E.9.2. Modificações

- Repair possible failure to update hint bits on disk  
Under rare circumstances this oversight could lead to “could not access transaction status” failures, which qualifies it as a potential-data-loss bug.
- Ensure that hashed outer join does not miss tuples  
Very large left joins using a hash join plan could fail to output unmatched left-side rows given just the right data distribution.
- Disallow running pg\_ctl as root  
This is to guard against any possible security issues.
- Avoid using temp files in /tmp in make\_oidjoins\_check  
This has been reported as a security issue, though it's hardly worthy of concern since there is no reason for non-developers to use this script anyway.

## E.10. Release 7.3.7

**Release date:** 2004-08-16

This release contains one critical fix over 7.3.6, and some minor items.

### E.10.1. Migration to version 7.3.7

A dump/restore is not required for those running 7.3.X.

### E.10.2. Modificações

- Prevent possible loss of committed transactions during crash

Due to insufficient interlocking between transaction commit and checkpointing, it was possible for transactions committed just before the most recent checkpoint to be lost, in whole or in part, following a database crash and restart. This is a serious bug that has existed since PostgreSQL 7.1.

- Remove asymmetrical word processing in tsearch (Teodor)
- Properly schema-qualify function names when pg\_dump'ing a CAST

## E.11. Release 7.3.6

**Release date:** 2004-03-02

This release contains a variety of fixes from 7.3.5.

### E.11.1. Migration to version 7.3.6

A dump/restore is *not* required for those running 7.3.\*.

### E.11.2. Modificações

- Revert erroneous changes in rule permissions checking

A patch applied in 7.3.3 to fix a corner case in rule permissions checks turns out to have disabled rule-related permissions checks in many not-so-corner cases. This would for example allow users to insert into views they weren't supposed to have permission to insert into. We have therefore reverted the 7.3.3 patch. The original bug will be fixed in 8.0.

- Repair incorrect order of operations in GetNewTransactionId()

This bug could result in failure under out-of-disk-space conditions, including inability to restart even after disk space is freed.

- Ensure configure selects -fno-strict-aliasing even when an external value for CFLAGS is supplied

On some platforms, building with -fstrict-aliasing causes bugs.

- Make pg\_restore handle 64-bit off\_t correctly

This bug prevented proper restoration from archive files exceeding 4Gb.

- Make contrib/dblink not assume that local and remote type OIDs match (Joe)
- Quote connectby()'s start\_with argument properly (Joe)
- Don't crash when a rowtype argument to a plpgsql function is NULL
- Avoid generating invalid character encoding sequences in corner cases when planning LIKE operations
- Ensure text\_position() cannot scan past end of source string in multibyte cases (Korea PostgreSQL Users' Group)
- Fix index optimization and selectivity estimates for LIKE operations on bytea columns (Joe)

## E.12. Release 7.3.5

**Release date:** 2003-12-03

This has a variety of fixes from 7.3.4.

### E.12.1. Migration to version 7.3.5

A dump/restore is *not* required for those running 7.3.\*.

## E.12.2. Modificações

- Force zero\_damaged\_pages to be on during recovery from WAL
- Prevent some obscure cases of “variable not in subplan target lists”
- Force stats processes to detach from shared memory, ensuring cleaner shutdown
- Make PQescapeBytea and byteaout consistent with each other (Joe)
- Added missing SPI\_finish() calls to dblink's get\_tuple\_of\_interest() (Joe)
- Fix for possible foreign key violation when rule rewrites INSERT (Jan)
- Support qualified type names in PL/Tcl's spi\_prepare command (Jan)
- Make pg\_dump handle a procedural language handler located in pg\_catalog
- Make pg\_dump handle cases where a custom opclass is in another schema
- Make pg\_dump dump binary-compatible casts correctly (Jan)
- Fix insertion of expressions containing subqueries into rule bodies
- Fix incorrect argument processing in clusterdb script (Anand Ranganathan)
- Fix problems with dropped columns in plpython triggers
- Repair problems with to\_char() reading past end of its input string (Karel)
- Fix GB18030 mapping errors (Tatsuo)
- Fix several problems with SSL error handling and asynchronous SSL I/O
- Remove ability to bind a list of values to a single parameter in JDBC (prevents possible SQL-injection attacks)
- Fix some errors in HAVE\_INT64\_TIMESTAMP code paths
- Fix corner case for btree search in parallel with first root page split

## E.13. Release 7.3.4

**Release date:** 2003-07-24

This has a variety of fixes from 7.3.3.

### E.13.1. Migration to version 7.3.4

A dump/restore is *not* required for those running 7.3.\*.

### E.13.2. Modificações

- Repair breakage in timestamp-to-date conversion for dates before 2000
- Prevent rare possibility of server startup failure (Tom)
- Fix bugs in interval-to-time conversion (Tom)
- Add constraint names in a few places in pg\_dump (Rod)
- Improve performance of functions with many parameters (Tom)
- Fix to\_ascii() buffer overruns (Tom)
- Prevent restore of database comments from throwing an error (Tom)
- Work around buggy strxfrm() present in some Solaris releases (Tom)
- Properly escape jdbc setObject() strings to improve security (Barry)

## E.14. Release 7.3.3

**Release date:** 2003-05-22

This release contains a variety of fixes for version 7.3.2.

### E.14.1. Migration to version 7.3.3

A dump/restore is *not* required for those running version 7.3.\*.

### E.14.2. Modificações

- Repair sometimes-incorrect computation of StartUpID after a crash
- Avoid slowness with lots of deferred triggers in one transaction (Stephan)
- Don't lock referenced row when UPDATE doesn't change foreign key's value (Jan)
- Use `-fPIC` not `-fpic` on Sparc (Tom Callaway)
- Repair lack of schema-awareness in contrib/reindexdb
- Fix contrib/intarray error for zero-element result array (Teodor)
- Ensure createuser script will exit on control-C (Oliver)
- Fix errors when the type of a dropped column has itself been dropped
- CHECKPOINT does not cause database panic on failure in noncritical steps
- Accept 60 in seconds fields of timestamp, time, interval input values
- Issue notice, not error, if TIMESTAMP, TIME, or INTERVAL precision too large
- Fix abstime-to-time cast function (fix is not applied unless you initdb)
- Fix pg\_proc entry for timestamp\_tz\_izone (fix is not applied unless you initdb)
- Make EXTRACT(EPOCH FROM timestamp without time zone) treat input as local time
- 'now'::timestamp\_tz gave wrong answer if timezone changed earlier in transaction
- HAVE\_INT64\_TIMESTAMP code for time with timezone overwrote its input
- Accept GLOBAL TEMP/TEMPORARY as a synonym for TEMPORARY
- Avoid improper schema-privilege-check failure in foreign-key triggers
- Fix bugs in foreign-key triggers for SET DEFAULT action
- Fix incorrect time-qual check in row fetch for UPDATE and DELETE triggers
- Foreign-key clauses were parsed but ignored in ALTER TABLE ADD COLUMN
- Fix createlang script breakage for case where handler function already exists
- Fix misbehavior on zero-column tables in pg\_dump, COPY, ANALYZE, other places
- Fix misbehavior of func\_error() on type names containing '%'
- Fix misbehavior of replace() on strings containing '%'
- Regular-expression patterns containing certain multibyte characters failed
- Account correctly for NULLs in more cases in join size estimation
- Avoid conflict with system definition of isblank() function or macro
- Fix failure to convert large code point values in EUC\_TW conversions (Tatsuo)
- Fix error recovery for SSL\_read/SSL\_write calls
- Don't do early constant-folding of type coercion expressions
- Validate page header fields immediately after reading in any page
- Repair incorrect check for ungrouped variables in unnamed joins
- Fix buffer overrun in to\_ascii (Guido Notari)
- contrib/ltree fixes (Teodor)

- Fix core dump in deadlock detection on machines where char is unsigned
- Avoid running out of buffers in many-way indexscan (bug introduced in 7.3)
- Fix planner's selectivity estimation functions to handle domains properly
- Fix dbmirror memory-allocation bug (Steven Singer)
- Prevent infinite loop in `ln(numeric)` due to roundoff error
- `GROUP BY` got confused if there were multiple equal `GROUP BY` items
- Fix bad plan when inherited `UPDATE/DELETE` references another inherited table
- Prevent clustering on incomplete (partial or non-NULL-storing) indexes
- Service shutdown request at proper time if it arrives while still starting up
- Fix left-links in temporary indexes (could make backwards scans miss entries)
- Fix incorrect handling of `client_encoding` setting in `postgresql.conf` (Tatsuo)
- Fix failure to respond to `pg_ctl stop -m fast` after `Async_NotifyHandler` runs
- Fix SPI for case where rule contains multiple statements of the same type
- Fix problem with checking for wrong type of access privilege in rule query
- Fix problem with `EXCEPT` in `CREATE RULE`
- Prevent problem with dropping temp tables having serial columns
- Fix `replace_vars_with_subplan_refs` failure in complex views
- Fix regexp slowness in single-byte encodings (Tatsuo)
- Allow qualified type names in `CREATE CAST` and `DROP CAST`
- Accept `SETOF type[]`, which formerly had to be written `SETOF _type`
- Fix `pg_dump` core dump in some cases with procedural languages
- Force ISO datestyle in `pg_dump` output, for portability (Oliver)
- `pg_dump` failed to handle error return from `lo_read` (Oleg Drokin)
- `pg_dumpall` failed with groups having no members (Nick Eskelinen)
- `pg_dumpall` failed to recognize `--globals-only` switch
- `pg_restore` failed to restore blobs if `-X disable-triggers` is specified
- Repair intrafunction memory leak in `plpgsql`
- `pltcl's` `elog` command dumped core if given wrong parameters (Ian Harding)
- `plpython` used wrong value of `atttypmod` (Brad McLean)
- Fix improper quoting of boolean values in Python interface (D'Arcy)
- Added `addDataType()` method to `PGConnection` interface for JDBC
- Fixed various problems with updateable `ResultSets` for JDBC (Shawn Green)
- Fixed various problems with `DatabaseMetaData` for JDBC (Kris Jurka, Peter Royal)
- Fixed problem with parsing table ACLs in JDBC
- Better error message for character set conversion problems in JDBC

## E.15. Release 7.3.2

**Release date:** 2003-02-04

This release contains a variety of fixes for version 7.3.1.

### E.15.1. Migration to version 7.3.2

A dump/restore is *not* required for those running version 7.3.\*.

### E.15.2. Modificações

- Restore creation of OID column in CREATE TABLE AS / SELECT INTO
- Fix pg\_dump core dump when dumping views having comments
- Dump DEFERRABLE/INITIALLY DEFERRED constraints properly
- Fix UPDATE when child table's column numbering differs from parent
- Increase default value of max\_fsm\_relations
- Fix problem when fetching backwards in a cursor for a single-row query
- Make backward fetch work properly with cursor on SELECT DISTINCT query
- Fix problems with loading pg\_dump files containing contrib/lo usage
- Fix problem with all-numeric user names
- Fix possible memory leak and core dump during disconnect in libpgtcl
- Make plpython's spi\_execute command handle nulls properly (Andrew Bosma)
- Adjust plpython error reporting so that its regression test passes again
- Work with bison 1.875
- Handle mixed-case names properly in plpgsql's %type (Neil)
- Fix core dump in pltcl when executing a query rewritten by a rule
- Repair array subscript overruns (per report from Yichen Xie)
- Reduce MAX\_TIME\_PRECISION from 13 to 10 in floating-point case
- Correctly case-fold variable names in per-database and per-user settings
- Fix coredump in plpgsql's RETURN NEXT when SELECT into record returns no rows
- Fix outdated use of pg\_type.typprtlen in python client interface
- Correctly handle fractional seconds in timestamps in JDBC driver
- Improve performance of getImportedKeys() in JDBC
- Make shared-library symlinks work standardly on HP-UX (Giles)
- Repair inconsistent rounding behavior for timestamp, time, interval
- SSL negotiation fixes (Nathan Mueller)
- Make libpq's ~/.pgpass feature work when connecting with PQconnectDB
- Update my2pg, ora2pg
- Translation updates
- Add casts between types lo and oid in contrib/lo
- fastpath code now checks for privilege to call function

### E.16. Release 7.3.1

**Release date:** 2002-12-18

This release contains a variety of fixes for version 7.3.



### E.16.1. Migration to version 7.3.1

A dump/restore is *not* required for those running version 7.3. However, it should be noted that the main PostgreSQL interface library, libpq, has a new major version number for this release, which may require recompilation of client code in certain cases.

### E.16.2. Modificações

- Fix a core dump of COPY TO when client/server encodings don't match (Tom)
- Allow pg\_dump to work with pre-7.2 servers (Philip)
- contrib/adddepend fixes (Tom)
- Fix problem with deletion of per-user/per-database config settings (Tom)
- contrib/vacuumlo fix (Tom)
- Allow 'password' encryption even when pg\_shadow contains MD5 passwords (Bruce)
- contrib/dbmirror fix (Steven Singer)
- Optimizer fixes (Tom)
- contrib/tsearch fixes (Teodor Sigaev, Magnus)
- Allow locale names to be mixed case (Nicolai Tufar)
- Increment libpq library's major version number (Bruce)
- pg\_hba.conf error reporting fixes (Bruce, Neil)
- Add SCO Openserver 5.0.4 as a supported platform (Bruce)
- Prevent EXPLAIN from crashing server (Tom)
- SSL fixes (Nathan Mueller)
- Prevent composite column creation via ALTER TABLE (Tom)

## E.17. Release 7.3

**Release date:** 2002-11-27

### E.17.1. Visão geral

Major changes in this release:

#### Schemas

Schemas allow users to create objects in separate namespaces, so two people or applications can have tables with the same name. There is also a public schema for shared tables. Table/index creation can be restricted by removing privileges on the public schema.

#### Drop Column

PostgreSQL now supports the `ALTER TABLE ... DROP COLUMN` functionality.

#### Table Functions

Functions returning multiple rows and/or multiple columns are now much easier to use than before. You can call such a “table function” in the `SELECT FROM` clause, treating its output like a table. Also, PL/pgSQL functions can now return sets.

#### Prepared Queries

PostgreSQL now supports prepared queries, for improved performance.

#### Dependency Tracking

PostgreSQL now records object dependencies, which allows improvements in many areas. `DROP` statements now take either `CASCADE` or `RESTRICT` to control whether dependent objects are also dropped.

## Privileges

Functions and procedural languages now have privileges, and functions can be defined to run with the privileges of their creator.

## Internationalization

Both multibyte and locale support are now always enabled.

## Logging

A variety of logging options have been enhanced.

## Interfaces

A large number of interfaces have been moved to <http://gborg.postgresql.org> where they can be developed and released independently.

## Functions/Identifiers

By default, functions can now take up to 32 parameters, and identifiers can be up to 63 bytes long. Also, `OPAQUE` is now deprecated: there are specific “pseudo-datatypes” to represent each of the former meanings of `OPAQUE` in function argument and result types.

## E.17.2. Migration to version 7.3

A dump/restore using `pg_dump` is required for those wishing to migrate data from any previous release. If your application examines the system catalogs, additional changes will be required due to the introduction of schemas in 7.3; for more information, see: [http://developer.postgresql.org/~momjian/upgrade\\_tips\\_7.3](http://developer.postgresql.org/~momjian/upgrade_tips_7.3) ([http://developer.postgresql.org/~momjian/upgrade\\_tips\\_7.3](http://developer.postgresql.org/~momjian/upgrade_tips_7.3)).

Observe the following incompatibilities:

- Pre-6.3 clients are no longer supported.
- `pg_hba.conf` now has a column for the user name and additional features. Existing files need to be adjusted.
- Several `postgresql.conf` logging parameters have been renamed.
- `LIMIT #, #` has been disabled; use `LIMIT # OFFSET #`.
- `INSERT` statements with column lists must specify a value for each specified column. For example, `INSERT INTO tab (col1, col2) VALUES ('val1')` is now invalid. It's still allowed to supply fewer columns than expected if the `INSERT` does not have a column list.
- `serial` columns are no longer automatically `UNIQUE`; thus, an index will not automatically be created.
- A `SET` command inside an aborted transaction is now rolled back.
- `COPY` no longer considers missing trailing columns to be null. All columns need to be specified. (However, one may achieve a similar effect by specifying a column list in the `COPY` command.)
- The data type `timestamp` is now equivalent to `timestamp without time zone`, instead of `timestamp with time zone`.
- Pre-7.3 databases loaded into 7.3 will not have the new object dependencies for `serial` columns, unique constraints, and foreign keys. See the directory `contrib/adddepend/` for a detailed description and a script that will add such dependencies.
- An empty string (`' '`) is no longer allowed as the input into an integer field. Formerly, it was silently interpreted as 0.

## E.17.3. Modificações

### E.17.3.1. Server Operation

- Add `pg_locks` view to show locks (Neil)
- Security fixes for password negotiation memory allocation (Neil)

- Remove support for version 0 FE/BE protocol (PostgreSQL 6.2 and earlier) (Tom)
- Reserve the last few backend slots for superusers, add parameter `superuser_reserved_connections` to control this (Nigel J. Andrews)

### **E.17.3.2. Performance**

- Improve startup by calling `localtime()` only once (Tom)
- Cache system catalog information in flat files for faster startup (Tom)
- Improve caching of index information (Tom)
- Optimizer improvements (Tom, Fernando Nasser)
- Catalog caches now store failed lookups (Tom)
- Hash function improvements (Neil)
- Improve performance of query tokenization and network handling (Peter)
- Speed improvement for large object restore (Mario Weilguni)
- Mark expired index entries on first lookup, saving later heap fetches (Tom)
- Avoid excessive NULL bitmap padding (Manfred Koizar)
- Add BSD-licensed `qsort()` for Solaris, for performance (Bruce)
- Reduce per-row overhead by four bytes (Manfred Koizar)
- Fix GEQO optimizer bug (Neil Conway)
- Make WITHOUT OID actually save four bytes per row (Manfred Koizar)
- Add `default_statistics_target` variable to specify ANALYZE buckets (Neil)
- Use local buffer cache for temporary tables so no WAL overhead (Tom)
- Improve free space map performance on large tables (Stephen Marshall, Tom)
- Improved WAL write concurrency (Tom)

### **E.17.3.3. Privilégios**

- Add privileges on functions and procedural languages (Peter)
- Add OWNER to CREATE DATABASE so superusers can create databases on behalf of unprivileged users (Gavin Sherry, Tom)
- Add new object privilege bits EXECUTE and USAGE (Tom)
- Add SET SESSION AUTHORIZATION DEFAULT and RESET SESSION AUTHORIZATION (Tom)
- Allow functions to be executed with the privilege of the function owner (Peter)

### **E.17.3.4. Server Configuration**

- Server log messages now tagged with LOG, not DEBUG (Bruce)
- Add user column to `pg_hba.conf` (Bruce)
- Have `log_connections` output two lines in log file (Tom)
- Remove `debug_level` from `postgresql.conf`, now `server_min_messages` (Bruce)
- New ALTER DATABASE/USER ... SET command for per-user/database initialization (Peter)
- New parameters `server_min_messages` and `client_min_messages` to control which messages are sent to the server logs or client applications (Bruce)
- Allow `pg_hba.conf` to specify lists of users/databases separated by commas, group names prepended with `+`, and file names prepended with `@` (Bruce)
- Remove secondary password file capability and `pg_password` utility (Bruce)
- Add variable `db_user_namespace` for database-local user names (Bruce)

- SSL improvements (Bear Giles)
- Make encryption of stored passwords the default (Bruce)
- Allow `pg_statistics` to be reset by calling `pg_stat_reset()` (Christopher)
- Add `log_duration` parameter (Bruce)
- Rename `debug_print_query` to `log_statement` (Bruce)
- Rename `show_query_stats` to `show_statement_stats` (Bruce)
- Add param `log_min_error_statement` to print commands to logs on error (Gavin)

#### **E.17.3.5. Consultas**

- Make cursors insensitive, meaning their contents do not change (Tom)
- Disable `LIMIT #, #` syntax; now only `LIMIT # OFFSET #` supported (Bruce)
- Increase identifier length to 63 (Neil, Bruce)
- UNION fixes for merging  $\geq 3$  columns of different lengths (Tom)
- Add `DEFAULT` key word to `INSERT`, e.g., `INSERT ... (... , DEFAULT, ...)` (Rod)
- Allow views to have default values using `ALTER COLUMN ... SET DEFAULT` (Neil)
- Fail on `INSERT`s with column lists that don't supply all column values, e.g., `INSERT INTO tab (col1, col2) VALUES ('val1');` (Rod)
- Fix for join aliases (Tom)
- Fix for `FULL OUTER JOINS` (Tom)
- Improve reporting of invalid identifier and location (Tom, Gavin)
- Fix `OPEN cursor(args)` (Tom)
- Allow 'ctid' to be used in a view and `currtd(viewname)` (Hiroshi)
- Fix for `CREATE TABLE AS` with `UNION` (Tom)
- SQL99 syntax improvements (Thomas)
- Add `statement_timeout` variable to cancel queries (Bruce)
- Allow prepared queries with `PREPARE/EXECUTE` (Neil)
- Allow `FOR UPDATE` to appear after `LIMIT/OFFSET` (Bruce)
- Add variable `autocommit` (Tom, David Van Wie)

#### **E.17.3.6. Object Manipulation**

- Make equals signs optional in `CREATE DATABASE` (Gavin Sherry)
- Make `ALTER TABLE OWNER` change index ownership too (Neil)
- New `ALTER TABLE tablename ALTER COLUMN colname SET STORAGE` controls `TOAST` storage, compression (John Gray)
- Add schema support, `CREATE/DROP SCHEMA` (Tom)
- Create schema for temporary tables (Tom)
- Add variable `search_path` for schema search (Tom)
- Add `ALTER TABLE SET/DROP NOT NULL` (Christopher)
- New `CREATE FUNCTION` volatility levels (Tom)
- Make rule names unique only per table (Tom)
- Add 'ON tablename' clause to `DROP RULE` and `COMMENT ON RULE` (Tom)
- Add `ALTER TRIGGER RENAME` (Joe)

- New `current_schema()` and `current_schemas()` inquiry functions (Tom)
- Allow functions to return multiple rows (table functions) (Joe)
- Make `WITH` optional in `CREATE DATABASE`, for consistency (Bruce)
- Add object dependency tracking (Rod, Tom)
- Add `RESTRICT/CASCADE` to `DROP` commands (Rod)
- Add `ALTER TABLE DROP` for non-CHECK CONSTRAINT (Rod)
- Autodestroy sequence on `DROP` of table with `SERIAL` (Rod)
- Prevent column dropping if column is used by foreign key (Rod)
- Automatically drop constraints/functions when object is dropped (Rod)
- Add `CREATE/DROP OPERATOR CLASS` (Bill Studenmund, Tom)
- Add `ALTER TABLE DROP COLUMN` (Christopher, Tom, Hiroshi)
- Prevent inherited columns from being removed or renamed (Alvaro Herrera)
- Fix foreign key constraints to not error on intermediate database states (Stephan)
- Propagate column or table renaming to foreign key constraints
- Add `CREATE OR REPLACE VIEW` (Gavin, Neil, Tom)
- Add `CREATE OR REPLACE RULE` (Gavin, Neil, Tom)
- Have rules execute alphabetically, returning more predictable values (Tom)
- Triggers are now fired in alphabetical order (Tom)
- Add `/contrib/adddepend` to handle pre-7.3 object dependencies (Rod)
- Allow better casting when inserting/updating values (Tom)

#### **E.17.3.7. Utility Commands**

- Have `COPY TO` output embedded carriage returns and newlines as `\r` and `\n` (Tom)
- Allow `DELIMITER` in `COPY FROM` to be 8-bit clean (Tatsuo)
- Make `pg_dump` use `ALTER TABLE ADD PRIMARY KEY`, for performance (Neil)
- Disable brackets in multistatement rules (Bruce)
- Disable `VACUUM` from being called inside a function (Bruce)
- Allow `dropdb` and other scripts to use identifiers with spaces (Bruce)
- Restrict database comment changes to the current database
- Allow comments on operators, independent of the underlying function (Rod)
- Rollback `SET` commands in aborted transactions (Tom)
- `EXPLAIN` now outputs as a query (Tom)
- Display condition expressions and sort keys in `EXPLAIN` (Tom)
- Add `'SET LOCAL var = value'` to set configuration variables for a single transaction (Tom)
- Allow `ANALYZE` to run in a transaction (Bruce)
- Improve `COPY` syntax using new `WITH` clauses, keep backward compatibility (Bruce)
- Fix `pg_dump` to consistently output tags in non-ASCII dumps (Bruce)
- Make foreign key constraints clearer in dump file (Rod)
- Add `COMMENT ON CONSTRAINT` (Rod)
- Allow `COPY TO/FROM` to specify column names (Brent Verner)
- Dump `UNIQUE` and `PRIMARY KEY` constraints as `ALTER TABLE` (Rod)

- Have SHOW output a query result (Joe)
- Generate failure on short COPY lines rather than pad NULLs (Neil)
- Fix CLUSTER to preserve all table attributes (Alvaro Herrera)
- New pg\_settings table to view/modify GUC settings (Joe)
- Add smart quoting, portability improvements to pg\_dump output (Peter)
- Dump serial columns out as SERIAL (Tom)
- Enable large file support, >2G for pg\_dump (Peter, Philip Warner, Bruce)
- Disallow TRUNCATE on tables that are involved in referential constraints (Rod)
- Have TRUNCATE also auto-truncate the toast table of the relation (Tom)
- Add clusterdb utility that will auto-cluster an entire database based on previous CLUSTER operations (Alvaro Herrera)
- Overhaul pg\_dumpall (Peter)
- Allow REINDEX of TOAST tables (Tom)
- Implemented START TRANSACTION, per SQL99 (Neil)
- Fix rare index corruption when a page split affects bulk delete (Tom)
- Fix ALTER TABLE ... ADD COLUMN for inheritance (Alvaro Herrera)

#### **E.17.3.8. Data Types and Functions**

- Fix factorial(0) to return 1 (Bruce)
- Date/time/timezone improvements (Thomas)
- Fix for array slice extraction (Tom)
- Fix extract/date\_part to report proper microseconds for timestamp (Tatsuo)
- Allow text\_substr() and bytea\_substr() to read TOAST values more efficiently (John Gray)
- Add domain support (Rod)
- Make WITHOUT TIME ZONE the default for TIMESTAMP and TIME data types (Thomas)
- Allow alternate storage scheme of 64-bit integers for date/time types using --enable-integer-datetimes in configure (Thomas)
- Make timezone(timestamptz) return timestamp rather than a string (Thomas)
- Allow fractional seconds in date/time types for dates prior to 1BC (Thomas)
- Limit timestamp data types to 6 decimal places of precision (Thomas)
- Change timezone conversion functions from timetz() to timezone() (Thomas)
- Add configuration variables datestyle and timezone (Tom)
- Add OVERLAY(), which allows substitution of a substring in a string (Thomas)
- Add SIMILAR TO (Thomas, Tom)
- Add regular expression SUBSTRING(string FROM pat FOR escape) (Thomas)
- Add LOCALTIME and LOCALTIMESTAMP functions (Thomas)
- Add named composite types using CREATE TYPE typename AS (column) (Joe)
- Allow composite type definition in the table alias clause (Joe)
- Add new API to simplify creation of C language table functions (Joe)
- Remove ODBC-compatible empty parentheses from calls to SQL99 functions for which these parentheses do not match the standard (Thomas)
- Allow macaddr data type to accept 12 hex digits with no separators (Mike Wyer)
- Add CREATE/DROP CAST (Peter)

- Add IS DISTINCT FROM operator (Thomas)
- Add SQL99 TREAT() function, synonym for CAST() (Thomas)
- Add pg\_backend\_pid() to output backend pid (Bruce)
- Add IS OF / IS NOT OF type predicate (Thomas)
- Allow bit string constants without fully-specified length (Thomas)
- Allow conversion between 8-byte integers and bit strings (Thomas)
- Implement hex literal conversion to bit string literal (Thomas)
- Allow table functions to appear in the FROM clause (Joe)
- Increase maximum number of function parameters to 32 (Bruce)
- No longer automatically create index for SERIAL column (Tom)
- Add current\_database() (Rod)
- Fix cash\_words() to not overflow buffer (Tom)
- Add functions replace(), split\_part(), to\_hex() (Joe)
- Fix LIKE for bytea as a right-hand argument (Joe)
- Prevent crashes caused by SELECT cash\_out(2) (Tom)
- Fix to\_char(1,'FM999.99') to return a period (Karel)
- Fix trigger/type/language functions returning OPAQUE to return proper type (Tom)

#### **E.17.3.9. Internacionalização**

- Add additional encodings: Korean (JOHAB), Thai (WIN874), Vietnamese (TCVN), Arabic (WIN1256), Simplified Chinese (GBK), Korean (UHC) (Eiji Tokuya)
- Enable locale support by default (Peter)
- Add locale variables (Peter)
- Escape bytes  $\geq 0x7f$  for multibyte in PQescapeBytea/PQunescapeBytea (Tatsuo)
- Add locale awareness to regular expression character classes
- Enable multibyte support by default (Tatsuo)
- Add GB18030 multibyte support (Bill Huang)
- Add CREATE/DROP CONVERSION, allowing loadable encodings (Tatsuo, Kaori)
- Add pg\_conversion table (Tatsuo)
- Add SQL99 CONVERT() function (Tatsuo)
- pg\_dumpall, pg\_controldata, and pg\_resetxlog now national-language aware (Peter)
- New and updated translations

#### **E.17.3.10. Server-side Languages**

- Allow recursive SQL function (Peter)
- Change PL/Tcl build to use configured compiler and Makefile.shlib (Peter)
- Overhaul the PL/pgSQL FOUND variable to be more Oracle-compatible (Neil, Tom)
- Allow PL/pgSQL to handle quoted identifiers (Tom)
- Allow set-returning PL/pgSQL functions (Neil)
- Make PL/pgSQL schema-aware (Joe)
- Remove some memory leaks (Nigel J. Andrews, Tom)

### **E.17.3.11. psql**

- Don't lowercase psql \connect database name for 7.2.0 compatibility (Tom)
- Add psql \timing to time user queries (Greg Sabino Mullane)
- Have psql \d show index information (Greg Sabino Mullane)
- New psql \dD shows domains (Jonathan Eisler)
- Allow psql to show rules on views (Paul ?)
- Fix for psql variable substitution (Tom)
- Allow psql \d to show temporary table structure (Tom)
- Allow psql \d to show foreign keys (Rod)
- Fix \? to honor \pset pager (Bruce)
- Have psql reports its version number on startup (Tom)
- Allow \copy to specify column names (Tom)

### **E.17.3.12. libpq**

- Add ~/.pgpass to store host/user password combinations (Alvaro Herrera)
- Add PQunescapeBytea() function to libpq (Patrick Welche)
- Fix for sending large queries over non-blocking connections (Bernhard Herzog)
- Fix for libpq using timers on Win9X (David Ford)
- Allow libpq notify to handle servers with different-length identifiers (Tom)
- Add libpq PQescapeString() and PQescapeBytea() to Windows (Bruce)
- Fix for SSL with non-blocking connections (Jack Bates)
- Add libpq connection timeout parameter (Denis A Ustimenko)

### **E.17.3.13. JDBC**

- Allow JDBC to compile with JDK 1.4 (Dave)
- Add JDBC 3 support (Barry)
- Allows JDBC to set loglevel by adding ?loglevel=X to the connection URL (Barry)
- Add Driver.info() message that prints out the version number (Barry)
- Add updateable result sets (Raghu Nidagal, Dave)
- Add support for callable statements (Paul Bethe)
- Add query cancel capability
- Add refresh row (Dave)
- Fix MD5 encryption handling for multibyte servers (Jun Kawai)
- Add support for prepared statements (Barry)

### **E.17.3.14. Miscellaneous Interfaces**

- Fixed ECPG bug concerning octal numbers in single quotes (Michael)
- Move src/interfaces/libpqeasy to <http://gborg.postgresql.org> (Marc, Bruce)
- Improve Python interface (Elliot Lee, Andrew Johnson, Greg Copeland)
- Add libpgtcl connection close event (Gerhard Hintermayer)
- Move src/interfaces/libpq++ to <http://gborg.postgresql.org> (Marc, Bruce)
- Move src/interfaces/odbc to <http://gborg.postgresql.org> (Marc)



- Move `src/interfaces/libpgeasy` to <http://gborg.postgresql.org> (Marc, Bruce)
- Move `src/interfaces/perl5` to <http://gborg.postgresql.org> (Marc, Bruce)
- Remove `src/bin/pgaccess` from main tree, now at <http://www.pgaccess.org> (Bruce)
- Add `pg_on_connection_loss` command to `libpgtcl` (Gerhard Hintermayer, Tom)

### **E.17.3.15. Source Code**

- Fix for parallel make (Peter)
- AIX fixes for linking Tcl (Andreas Zeugswetter)
- Allow PL/Perl to build under Cygwin (Jason Tishler)
- Improve MIPS compiles (Peter, Oliver Elphick)
- Require Autoconf version 2.53 (Peter)
- Require readline and zlib by default in configure (Peter)
- Allow Solaris to use Intimate Shared Memory (ISM), for performance (Scott Brunza, P.J. Josh Rovero)
- Always enable syslog in compile, remove `--enable-syslog` option (Tatsuo)
- Always enable multibyte in compile, remove `--enable-multibyte` option (Tatsuo)
- Always enable locale in compile, remove `--enable-locale` option (Peter)
- Fix for Win9x DLL creation (Magnus Naeslund)
- Fix for `link()` usage by WAL code on Windows, BeOS (Jason Tishler)
- Add `sys/types.h` to `c.h`, remove from main files (Peter, Bruce)
- Fix AIX hang on SMP machines (Tomoyuki Nijima)
- AIX SMP hang fix (Tomoyuki Nijima)
- Fix pre-1970 date handling on newer glibc libraries (Tom)
- Fix PowerPC SMP locking (Tom)
- Prevent gcc `-ffast-math` from being used (Peter, Tom)
- Bison `>= 1.50` now required for developer builds
- Kerberos 5 support now builds with Heimdal (Peter)
- Add appendix in the User's Guide which lists SQL features (Thomas)
- Improve loadable module linking to use `RTLD_NOW` (Tom)
- New error levels WARNING, INFO, LOG, DEBUG[1-5] (Bruce)
- New `src/port` directory holds replaced libc functions (Peter, Bruce)
- New `pg_namespace` system catalog for schemas (Tom)
- Add `pg_class.relnamespace` for schemas (Tom)
- Add `pg_type.typnamespace` for schemas (Tom)
- Add `pg_proc.pronamespace` for schemas (Tom)
- Restructure aggregates to have `pg_proc` entries (Tom)
- System relations now have their own namespace, `pg_*` test not required (Fernando Nasser)
- Rename TOAST index names to be `*_index` rather than `*_idx` (Neil)
- Add namespaces for operators, opclasses (Tom)
- Add additional checks to server control file (Thomas)
- New Polish FAQ (Marcin Mazurek)
- Add Posix semaphore support (Tom)

- Document need for reindex (Bruce)
- Rename some internal identifiers to simplify Windows compile (Jan, Katherine Ward)
- Add documentation on computing disk space (Bruce)
- Remove KSQO from GUC (Bruce)
- Fix memory leak in rtree (Kenneth Been)
- Modify a few error messages for consistency (Bruce)
- Remove unused system table columns (Peter)
- Make system columns NOT NULL where appropriate (Tom)
- Clean up use of sprintf in favor of snprintf() (Neil, Jukka Holappa)
- Remove OPAQUE and create specific subtypes (Tom)
- Cleanups in array internal handling (Joe, Tom)
- Disallow pg\_atoi("") (Bruce)
- Remove parameter wal\_files because WAL files are now recycled (Bruce)
- Add version numbers to heap pages (Tom)

#### **E.17.3.16. Contrib**

- Allow inet arrays in /contrib/array (Neil)
- GiST fixes (Teodor Sigaev, Neil)
- Upgrade /contrib/mysql
- Add /contrib/dbsize which shows table sizes without vacuum (Peter)
- Add /contrib/intagg, integer aggregator routines (mlw)
- Improve /contrib/oid2name (Neil, Bruce)
- Improve /contrib/tsearch (Oleg, Teodor Sigaev)
- Cleanups of /contrib/rserver (Alexey V. Borzov)
- Update /contrib/oracle conversion utility (Gilles Darold)
- Update /contrib/dblink (Joe)
- Improve options supported by /contrib/vacuumlo (Mario Weilguni)
- Improvements to /contrib/intarray (Oleg, Teodor Sigaev, Andrey Oktyabrski)
- Add /contrib/reindexdb utility (Shaun Thomas)
- Add indexing to /contrib/isbn\_issn (Dan Weston)
- Add /contrib/dbmirror (Steven Singer)
- Improve /contrib/pgbench (Neil)
- Add /contrib/tablefunc table function examples (Joe)
- Add /contrib/ltree data type for tree structures (Teodor Sigaev, Oleg Bartunov)
- Move /contrib/pg\_controldata, pg\_resetxlog into main tree (Bruce)
- Fixes to /contrib/cube (Bruno Wolff)
- Improve /contrib/fulltextindex (Christopher)

### **E.18. Release 7.2.6**

**Release date:** 2004-10-22

This release contains a variety of fixes from 7.2.5.

### E.18.1. Migration to version 7.2.6

A dump/restore is not required for those running 7.2.X.

### E.18.2. Modificações

- Repair possible failure to update hint bits on disk  
Under rare circumstances this oversight could lead to “could not access transaction status” failures, which qualifies it as a potential-data-loss bug.
- Ensure that hashed outer join does not miss tuples  
Very large left joins using a hash join plan could fail to output unmatched left-side rows given just the right data distribution.
- Disallow running `pg_ctl` as root  
This is to guard against any possible security issues.
- Avoid using temp files in `/tmp` in `make_oidjoins_check`  
This has been reported as a security issue, though it's hardly worthy of concern since there is no reason for non-developers to use this script anyway.
- Update to newer versions of Bison

## E.19. Release 7.2.5

**Release date:** 2004-08-16

This release contains a variety of fixes from 7.2.4.

### E.19.1. Migration to version 7.2.5

A dump/restore is not required for those running 7.2.X.

### E.19.2. Modificações

- Prevent possible loss of committed transactions during crash  
Due to insufficient interlocking between transaction commit and checkpointing, it was possible for transactions committed just before the most recent checkpoint to be lost, in whole or in part, following a database crash and restart. This is a serious bug that has existed since PostgreSQL 7.1.
- Fix corner case for btree search in parallel with first root page split
- Fix buffer overrun in `to_ascii` (Guido Notari)
- Fix core dump in deadlock detection on machines where `char` is unsigned
- Fix failure to respond to `pg_ctl stop -m fast` after `Async_NotifyHandler` runs
- Repair memory leaks in `pg_dump`
- Avoid conflict with system definition of `isblank()` function or macro

## E.20. Release 7.2.4

**Release date:** 2003-01-30

This release contains a variety of fixes for version 7.2.3, including fixes to prevent possible data loss.

### E.20.1. Migration to version 7.2.4

A dump/restore is *not* required for those running version 7.2.\*.

## E.20.2. Modificações

- Fix some additional cases of VACUUM "No one parent tuple was found" error
- Prevent VACUUM from being called inside a function (Bruce)
- Ensure pg\_clog updates are sync'd to disk before marking checkpoint complete
- Avoid integer overflow during large hash joins
- Make GROUP commands work when pg\_group.grobject is large enough to be toasted
- Fix errors in datetime tables; some timezone names weren't being recognized
- Fix integer overflows in circle\_poly(), path\_encode(), path\_add() (Neil)
- Repair long-standing logic errors in lseg\_eq(), lseg\_ne(), lseg\_center()

## E.21. Release 7.2.3

**Release date:** 2002-10-01

This release contains a variety of fixes for version 7.2.2, including fixes to prevent possible data loss.

### E.21.1. Migration to version 7.2.3

A dump/restore is *not* required for those running version 7.2.\*.

### E.21.2. Modificações

- Prevent possible compressed transaction log loss (Tom)
- Prevent non-superuser from increasing most recent vacuum info (Tom)
- Handle pre-1970 date values in newer versions of glibc (Tom)
- Fix possible hang during server shutdown
- Prevent spinlock hangs on SMP PPC machines (Tomoyuki Nijima)
- Fix pg\_dump to properly dump FULL JOIN USING (Tom)

## E.22. Release 7.2.2

**Release date:** 2002-08-23

This release contains a variety of fixes for version 7.2.1.

### E.22.1. Migration to version 7.2.2

A dump/restore is *not* required for those running version 7.2.\*.

### E.22.2. Modificações

- Allow EXECUTE of "CREATE TABLE AS ... SELECT" in PL/pgSQL (Tom)
- Fix for compressed transaction log id wraparound (Tom)
- Fix PQescapeBytea/PQunescapeBytea so that they handle bytes > 0x7f (Tatsuo)
- Fix for psql and pg\_dump crashing when invoked with non-existent long options (Tatsuo)
- Fix crash when invoking geometric operators (Tom)
- Allow OPEN cursor(args) (Tom)
- Fix for rtree\_gist index build (Teodor)
- Fix for dumping user-defined aggregates (Tom)
- contrib/intarray fixes (Oleg)

- Fix for complex UNION/EXCEPT/INTERSECT queries using parens (Tom)
- Fix to pg\_convert (Tatsuo)
- Fix for crash with long DATA strings (Thomas, Neil)
- Fix for repeat(), lpad(), rpad() and long strings (Neil)

## E.23. Release 7.2.1

**Release date:** 2002-03-21

This release contains a variety of fixes for version 7.2.

### E.23.1. Migration to version 7.2.1

A dump/restore is *not* required for those running version 7.2.

### E.23.2. Modificações

- Ensure that sequence counters do not go backwards after a crash (Tom)
- Fix pgaccess kanji-conversion key binding (Tatsuo)
- Optimizer improvements (Tom)
- Cash I/O improvements (Tom)
- New Russian FAQ
- Compile fix for missing AuthBlockSig (Heiko)
- Additional time zones and time zone fixes (Thomas)
- Allow psql \connect to handle mixed case database and user names (Tom)
- Return proper OID on command completion even with ON INSERT rules (Tom)
- Allow COPY FROM to use 8-bit DELIMITERS (Tatsuo)
- Fix bug in extract/date\_part for milliseconds/microseconds (Tatsuo)
- Improve handling of multiple UNIONS with different lengths (Tom)
- contrib/btree\_gist improvements (Teodor Sigaev)
- contrib/tsearch dictionary improvements, see README.tsearch for an additional installation step (Thomas T. Thai, Teodor Sigaev)
- Fix for array subscripts handling (Tom)
- Allow EXECUTE of "CREATE TABLE AS ... SELECT" in PL/pgSQL (Tom)

## E.24. Release 7.2

**Release date:** 2002-02-04

### E.24.1. Visão geral

This release improves PostgreSQL for use in high-volume applications.

Major changes in this release:

#### VACUUM

Vacuuming no longer locks tables, thus allowing normal user access during the vacuum. A new `VACUUM FULL` command does old-style vacuum by locking the table and shrinking the on-disk copy of the table.

#### Transactions

There is no longer a problem with installations that exceed four billion transactions.

## OIDs

OIDs are now optional. Users can now create tables without OIDs for cases where OID usage is excessive.

## Optimizer

The system now computes histogram column statistics during `ANALYZE`, allowing much better optimizer choices.

## Security

A new MD5 encryption option allows more secure storage and transfer of passwords. A new Unix-domain socket authentication option is available on Linux and BSD systems.

## Statistics

Administrators can use the new table access statistics module to get fine-grained information about table and index usage.

## Internationalization

Program and library messages can now be displayed in several languages.

## E.24.2. Migration to version 7.2

A dump/restore using `pg_dump` is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

- The semantics of the `VACUUM` command have changed in this release. You may wish to update your maintenance procedures accordingly.
- In this release, comparisons using `= NULL` will always return false (or `NULL`, more precisely). Previous releases automatically transformed this syntax to `IS NULL`. The old behavior can be re-enabled using a `postgresql.conf` parameter.
- The `pg_hba.conf` and `pg_ident.conf` configuration is now only reloaded after receiving a `SIGHUP` signal, not with each connection.
- The function `octet_length()` now returns the uncompressed data length.
- The date/time value 'current' is no longer available. You will need to rewrite your applications.
- The `timestamp()`, `time()`, and `interval()` functions are no longer available. Instead of `timestamp()`, use `timestamp 'string' or CAST`.

The `SELECT ... LIMIT #, #` syntax will be removed in the next release. You should change your queries to use separate `LIMIT` and `OFFSET` clauses, e.g. `LIMIT 10 OFFSET 20`.

## E.24.3. Modificações

### E.24.3.1. Server Operation

- Create temporary files in a separate directory (Bruce)
- Delete orphaned temporary files on postmaster startup (Bruce)
- Added unique indexes to some system tables (Tom)
- System table operator reorganization (Oleg Bartunov, Teodor Sigaev, Tom)
- Renamed `pg_log` to `pg_clog` (Tom)
- Enable `SIGTERM`, `SIGQUIT` to kill backends (Jan)
- Removed compile-time limit on number of backends (Tom)
- Better cleanup for semaphore resource failure (Tatsuo, Tom)
- Allow safe transaction ID wraparound (Tom)

- Removed OIDs from some system tables (Tom)
- Removed "triggered data change violation" error check (Tom)
- SPI portal creation of prepared/saved plans (Jan)
- Allow SPI column functions to work for system columns (Tom)
- Long value compression improvement (Tom)
- Statistics collector for table, index access (Jan)
- Truncate extra-long sequence names to a reasonable value (Tom)
- Measure transaction times in milliseconds (Thomas)
- Fix TID sequential scans (Hiroshi)
- Superuser ID now fixed at 1 (Peter E)
- New pg\_ctl "reload" option (Tom)

#### **E.24.3.2. Performance**

- Optimizer improvements (Tom)
- New histogram column statistics for optimizer (Tom)
- Reuse write-ahead log files rather than discarding them (Tom)
- Cache improvements (Tom)
- IS NULL, IS NOT NULL optimizer improvement (Tom)
- Improve lock manager to reduce lock contention (Tom)
- Keep relcache entries for index access support functions (Tom)
- Allow better selectivity with NaN and infinities in NUMERIC (Tom)
- R-tree performance improvements (Kenneth Been)
- B-tree splits more efficient (Tom)

#### **E.24.3.3. Privilégios**

- Change UPDATE, DELETE privileges to be distinct (Peter E)
- New REFERENCES, TRIGGER privileges (Peter E)
- Allow GRANT/REVOKE to/from more than one user at a time (Peter E)
- New has\_table\_privilege() function (Joe Conway)
- Allow non-superuser to vacuum database (Tom)
- New SET SESSION AUTHORIZATION command (Peter E)
- Fix bug in privilege modifications on newly created tables (Tom)
- Disallow access to pg\_statistic for non-superuser, add user-accessible views (Tom)

#### **E.24.3.4. Client Authentication**

- Fork postmaster before doing authentication to prevent hangs (Peter E)
- Add ident authentication over Unix domain sockets on Linux, \*BSD (Helge Bahmann, Oliver Elphick, Teodor Sigaev, Bruce)
- Add a password authentication method that uses MD5 encryption (Bruce)
- Allow encryption of stored passwords using MD5 (Bruce)
- PAM authentication (Dominic J. Eidson)
- Load pg\_hba.conf and pg\_ident.conf only on startup and SIGHUP (Bruce)

**E.24.3.5. Server Configuration**

- Interpretation of some time zone abbreviations as Australian rather than North American now settable at run time (Bruce)
- New parameter to set default transaction isolation level (Peter E)
- New parameter to enable conversion of "expr = NULL" into "expr IS NULL", off by default (Peter E)
- New parameter to control memory usage by VACUUM (Tom)
- New parameter to set client authentication timeout (Tom)
- New parameter to set maximum number of open files (Tom)

**E.24.3.6. Consultas**

- Statements added by INSERT rules now execute after the INSERT (Jan)
- Prevent unadorned relation names in target list (Bruce)
- NULLs now sort after all normal values in ORDER BY (Tom)
- New IS UNKNOWN, IS NOT UNKNOWN Boolean tests (Tom)
- New SHARE UPDATE EXCLUSIVE lock mode (Tom)
- New EXPLAIN ANALYZE command that shows run times and row counts (Martijn van Oosterhout)
- Fix problem with LIMIT and subqueries (Tom)
- Fix for LIMIT, DISTINCT ON pushed into subqueries (Tom)
- Fix nested EXCEPT/INTERSECT (Tom)

**E.24.3.7. Schema Manipulation**

- Fix SERIAL in temporary tables (Bruce)
- Allow temporary sequences (Bruce)
- Sequences now use int8 internally (Tom)
- New SERIAL8 creates int8 columns with sequences, default still SERIAL4 (Tom)
- Make OIDs optional using WITHOUT OIDS (Tom)
- Add %TYPE syntax to CREATE TYPE (Ian Lance Taylor)
- Add ALTER TABLE / DROP CONSTRAINT for CHECK constraints (Christopher Kings-Lynne)
- New CREATE OR REPLACE FUNCTION to alter existing function (preserving the function OID) (Gavin Sherry)
- Add ALTER TABLE / ADD [ UNIQUE | PRIMARY ] (Christopher Kings-Lynne)
- Allow column renaming in views
- Make ALTER TABLE / RENAME COLUMN update column names of indexes (Brent Verner)
- Fix for ALTER TABLE / ADD CONSTRAINT ... CHECK with inherited tables (Stephan Szabo)
- ALTER TABLE RENAME update foreign-key trigger arguments correctly (Brent Verner)
- DROP AGGREGATE and COMMENT ON AGGREGATE now accept an aggtype (Tom)
- Add automatic return type data casting for SQL functions (Tom)
- Allow GiST indexes to handle NULLs and multikey indexes (Oleg Bartunov, Teodor Sigaev, Tom)
- Enable partial indexes (Martijn van Oosterhout)

**E.24.3.8. Utility Commands**

- Add RESET ALL, SHOW ALL (Marko Kreen)
- CREATE/ALTER USER/GROUP now allow options in any order (Vince)
- Add LOCK A, B, C functionality (Neil Padgett)



- New ENCRYPTED/UNENCRYPTED option to CREATE/ALTER USER (Bruce)
- New light-weight VACUUM does not lock table; old semantics are available as VACUUM FULL (Tom)
- Disable COPY TO/FROM on views (Bruce)
- COPY DELIMITERS string must be exactly one character (Tom)
- VACUUM warning about index tuples fewer than heap now only appears when appropriate (Martijn van Oosterhout)
- Fix privilege checks for CREATE INDEX (Tom)
- Disallow inappropriate use of CREATE/DROP INDEX/TRIGGER/VIEW (Tom)

#### **E.24.3.9. Data Types and Functions**

- SUM(), AVG(), COUNT() now uses int8 internally for speed (Tom)
- Add convert(), convert2() (Tatsuo)
- New function bit\_length() (Peter E)
- Make the "n" in CHAR(n)/VARCHAR(n) represents letters, not bytes (Tatsuo)
- CHAR(), VARCHAR() now reject strings that are too long (Peter E)
- BIT VARYING now rejects bit strings that are too long (Peter E)
- BIT now rejects bit strings that do not match declared size (Peter E)
- INET, CIDR text conversion functions (Alex Pilosov)
- INET, CIDR operators << and <=< indexable (Alex Pilosov)
- Bytea \### now requires valid three digit octal number
- Bytea comparison improvements, now supports =, <>, >, >=, <, and <=
- Bytea now supports B-tree indexes
- Bytea now supports LIKE, LIKE...ESCAPE, NOT LIKE, NOT LIKE...ESCAPE
- Bytea now supports concatenation
- New bytea functions: position, substring, trim, btrim, and length
- New encode() function mode, "escaped", converts minimally escaped bytea to/from text
- Add pg\_database\_encoding\_max\_length() (Tatsuo)
- Add pg\_client\_encoding() function (Tatsuo)
- now() returns time with millisecond precision (Thomas)
- New TIMESTAMP WITHOUT TIMEZONE data type (Thomas)
- Add ISO date/time specification with "T", yyyy-mm-ddThh:mm:ss (Thomas)
- New xid/int comparison functions (Hiroshi)
- Add precision to TIME, TIMESTAMP, and INTERVAL data types (Thomas)
- Modify type coercion logic to attempt binary-compatible functions first (Tom)
- New encode() function installed by default (Marko Kreen)
- Improved to\_\*() conversion functions (Karel Zak)
- Optimize LIKE/ILIKE when using single-byte encodings (Tatsuo)
- New functions in contrib/pgcrypto: crypt(), hmac(), encrypt(), gen\_salt() (Marko Kreen)
- Correct description of translate() function (Bruce)
- Add INTERVAL argument for SET TIME ZONE (Thomas)
- Add INTERVAL YEAR TO MONTH (etc.) syntax (Thomas)
- Optimize length functions when using single-byte encodings (Tatsuo)

- Fix path\_inter, path\_distance, path\_length, dist\_ppath to handle closed paths (Curtis Barrett, Tom)
- octet\_length(text) now returns non-compressed length (Tatsuo, Bruce)
- Handle "July" full name in date/time literals (Greg Sabino Mullane)
- Some datatype() function calls now evaluated differently
- Add support for Julian and ISO time specifications (Thomas)

#### **E.24.3.10. Internacionalização**

- National language support in psql, pg\_dump, libpq, and server (Peter E)
- Message translations in Chinese (simplified, traditional), Czech, French, German, Hungarian, Russian, Swedish (Peter E, Serguei A. Mokhov, Karel Zak, Weiping He, Zhenbang Wei, Kovacs Zoltan)
- Make trim, ltrim, rtrim, btrim, lpad, rpad, translate multibyte aware (Tatsuo)
- Add LATIN5,6,7,8,9,10 support (Tatsuo)
- Add ISO 8859-5,6,7,8 support (Tatsuo)
- Correct LATIN5 to mean ISO-8859-9, not ISO-8859-5 (Tatsuo)
- Make mic2ascii() non-ASCII aware (Tatsuo)
- Reject invalid multibyte character sequences (Tatsuo)

#### **E.24.3.11. PL/pgSQL**

- Now uses portals for SELECT loops, allowing huge result sets (Jan)
- CURSOR and REFCURSOR support (Jan)
- Can now return open cursors (Jan)
- Add ELSEIF (Klaus Reger)
- Improve PL/pgSQL error reporting, including location of error (Tom)
- Allow IS or FOR key words in cursor declaration, for compatibility (Bruce)
- Fix for SELECT ... FOR UPDATE (Tom)
- Fix for PERFORM returning multiple rows (Tom)
- Make PL/pgSQL use the server's type coercion code (Tom)
- Memory leak fix (Jan, Tom)
- Make trailing semicolon optional (Tom)

#### **E.24.3.12. PL/Perl**

- New untrusted PL/Perl (Alex Pilosov)
- PL/Perl is now built on some platforms even if libperl is not shared (Peter E)

#### **E.24.3.13. PL/Tcl**

- Now reports errorInfo (Vsevolod Lobko)
- Add spi\_lastoid function (bob@redivi.com)

#### **E.24.3.14. PL/Python**

- ...is new (Andrew Bosma)

#### **E.24.3.15. psql**

- \d displays indexes in unique, primary groupings (Christopher Kings-Lynne)
- Allow trailing semicolons in backslash commands (Greg Sabino Mullane)
- Read password from /dev/tty if possible

- Force new password prompt when changing user and database (Tatsuo, Tom)
- Format the correct number of columns for Unicode (Patrice)

#### **E.24.3.16. libpq**

- New function PQescapeString() to escape quotes in command strings (Florian Weimer)
- New function PQescapeBytea() escapes binary strings for use as SQL string literals

#### **E.24.3.17. JDBC**

- Return OID of INSERT (Ken K)
- Handle more data types (Ken K)
- Handle single quotes and newlines in strings (Ken K)
- Handle NULL variables (Ken K)
- Fix for time zone handling (Barry Lind)
- Improved Druid support
- Allow eight-bit characters with non-multibyte server (Barry Lind)
- Support BIT, BINARY types (Ned Wolpert)
- Reduce memory usage (Michael Stephens, Dave Cramer)
- Update DatabaseMetaData (Peter E)
- Add DatabaseMetaData.getCatalogs() (Peter E)
- Encoding fixes (Anders Bengtsson)
- Get/setCatalog methods (Jason Davies)
- DatabaseMetaData.getColumns() now returns column defaults (Jason Davies)
- DatabaseMetaData.getColumns() performance improvement (Jeroen van Vianen)
- Some JDBC1 and JDBC2 merging (Anders Bengtsson)
- Transaction performance improvements (Barry Lind)
- Array fixes (Greg Zoller)
- Serialize addition
- Fix batch processing (Rene Pijlman)
- ExecSQL method reorganization (Anders Bengtsson)
- GetColumn() fixes (Jeroen van Vianen)
- Fix isWritable() function (Rene Pijlman)
- Improved passage of JDBC2 conformance tests (Rene Pijlman)
- Add bytea type capability (Barry Lind)
- Add isNullable() (Rene Pijlman)
- JDBC date/time test suite fixes (Liam Stewart)
- Fix for SELECT 'id' AS xxx FROM table (Dave Cramer)
- Fix DatabaseMetaData to show precision properly (Mark Lillywhite)
- New getImported/getExported keys (Jason Davies)
- MD5 password encryption support (Jeremy Wohl)
- Fix to actually use type cache (Ned Wolpert)

### **E.24.3.18. ODBC**

- Remove query size limit (Hiroshi)
- Remove text field size limit (Hiroshi)
- Fix for SQLPrimaryKeys in multibyte mode (Hiroshi)
- Allow ODBC procedure calls (Hiroshi)
- Improve boolean handing (Aidan Mountford)
- Most configuration options now settable via DSN (Hiroshi)
- Multibyte, performance fixes (Hiroshi)
- Allow driver to be used with iODBC or unixODBC (Peter E)
- MD5 password encryption support (Bruce)
- Add more compatibility functions to odbcc.sql (Peter E)

### **E.24.3.19. ECPG**

- EXECUTE ... INTO implemented (Christof Petig)
- Multiple row descriptor support (e.g. CARDINALITY) (Christof Petig)
- Fix for GRANT parameters (Lee Kindness)
- Fix INITIALLY DEFERRED bug
- Various bug fixes (Michael, Christof Petig)
- Auto allocation for indicator variable arrays (int \*ind\_p=NULL)
- Auto allocation for string arrays (char \*\*foo\_pp=NULL)
- ECPGfree\_auto\_mem fixed
- All function names with external linkage are now prefixed by ECPG
- Fixes for arrays of structures (Michael)

### **E.24.3.20. Misc. Interfaces**

- Python fix fetchone() (Gerhard Haring)
- Use UTF, Unicode in Tcl where appropriate (Vsevolod Lobko, Reinhard Max)
- Add Tcl COPY TO/FROM (ljb)
- Prevent output of default index op class in pg\_dump (Tom)
- Fix libpgeasy memory leak (Bruce)

### **E.24.3.21. Build and Install**

- Configure, dynamic loader, and shared library fixes (Peter E)
- Fixes in QNX 4 port (Bernd Tegge)
- Fixes in Cygwin and Windows ports (Jason Tishler, Gerhard Haring, Dmitry Yurtaev, Darko Prenosil, Mikhail Terekhov)
- Fix for Windows socket communication failures (Magnus, Mikhail Terekhov)
- Hurd compile fix (Oliver Elphick)
- BeOS fixes (Cyril Velter)
- Remove configure --enable-unicode-conversion, now enabled by multibyte (Tatsuo)
- AIX fixes (Tatsuo, Andreas)
- Fix parallel make (Peter E)
- Install SQL language manual pages into OS-specific directories (Peter E)

- Rename config.h to pg\_config.h (Peter E)
- Reorganize installation layout of header files (Peter E)

#### E.24.3.22. Source Code

- Remove SEP\_CHAR (Bruce)
- New GUC hooks (Tom)
- Merge GUC and command line handling (Marko Kreen)
- Remove EXTEND INDEX (Martijn van Oosterhout, Tom)
- New pgindent utility to indent java code (Bruce)
- Remove define of true/false when compiling under C++ (Leandro Fanzone, Tom)
- pgindent fixes (Bruce, Tom)
- Replace strcasecmp() with strcmp() where appropriate (Peter E)
- Dynahash portability improvements (Tom)
- Add 'volatile' usage in spinlock structures
- Improve signal handling logic (Tom)

#### E.24.3.23. Contrib

- New contrib/rtree\_gist (Oleg Bartunov, Teodor Sigaev)
- New contrib/tsearch full-text indexing (Oleg, Teodor Sigaev)
- Add contrib/dblink for remote database access (Joe Conway)
- contrib/ora2pg Oracle conversion utility (Gilles Darold)
- contrib/xml XML conversion utility (John Gray)
- contrib/fulltextindex fixes (Christopher Kings-Lynne)
- New contrib/fuzzystrmatch with levenshtein and metaphone, soundex merged (Joe Conway)
- Add contrib/intarray boolean queries, binary search, fixes (Oleg Bartunov)
- New pg\_upgrade utility (Bruce)
- Add new pg\_resetxlog options (Bruce, Tom)

## E.25. Release 7.1.3

**Release date:** 2001-08-15

### E.25.1. Migration to version 7.1.3

A dump/restore is *not* required for those running 7.1.X.

### E.25.2. Modificações

Remove unused WAL segments of large transactions (Tom)  
 Multiaction rule fix (Tom)  
 PL/pgSQL memory allocation fix (Jan)  
 VACUUM buffer fix (Tom)  
 Regression test fixes (Tom)  
 pg\_dump fixes for GRANT/REVOKE/comments on views, user-defined types (Tom)  
 Fix subselects with DISTINCT ON or LIMIT (Tom)  
 BeOS fix  
 Disable COPY TO/FROM a view (Tom)  
 Cygwin build (Jason Tishler)

## E.26. Release 7.1.2

**Release date:** 2001-05-11

This has one fix from 7.1.1.

### E.26.1. Migration to version 7.1.2

A dump/restore is *not* required for those running 7.1.X.

### E.26.2. Modificações

Fix PL/pgSQL SELECTs when returning no rows  
 Fix for psql backslash core dump  
 Referential integrity privilege fix  
 Optimizer fixes  
 pg\_dump cleanups

## E.27. Release 7.1.1

**Release date:** 2001-05-05

This has a variety of fixes from 7.1.

### E.27.1. Migration to version 7.1.1

A dump/restore is *not* required for those running 7.1.

### E.27.2. Modificações

Fix for numeric MODULO operator (Tom)  
 pg\_dump fixes (Philip)  
 pg\_dump can dump 7.0 databases (Philip)  
 readline 4.2 fixes (Peter E)  
 JOIN fixes (Tom)  
 AIX, MSWIN, VAX, N32K fixes (Tom)  
 Multibytes fixes (Tom)  
 Unicode fixes (Tatsuo)  
 Optimizer improvements (Tom)  
 Fix for whole rows in functions (Tom)  
 Fix for pg\_ctl and option strings with spaces (Peter E)  
 ODBC fixes (Hiroshi)  
 EXTRACT can now take string argument (Thomas)  
 Python fixes (Darcy)

## E.28. Release 7.1

**Release date:** 2001-04-13

This release focuses on removing limitations that have existed in the PostgreSQL code for many years.

Major changes in this release:

Write-ahead Log (WAL)

To maintain database consistency in case of an operating system crash, previous releases of PostgreSQL have forced all data modifications to disk before each transaction commit. With WAL, only one log file must be flushed to disk,

greatly improving performance. If you have been using `-F` in previous releases to disable disk flushes, you may want to consider discontinuing its use.

## TOAST

TOAST - Previous releases had a compiled-in row length limit, typically 8k - 32k. This limit made storage of long text fields difficult. With TOAST, long rows of any length can be stored with good performance.

## Outer Joins

We now support outer joins. The UNION/NOT IN workaround for outer joins is no longer required. We use the SQL92 outer join syntax.

## Function Manager

The previous C function manager did not handle null values properly, nor did it support 64-bit CPU's (Alpha). The new function manager does. You can continue using your old custom functions, but you may want to rewrite them in the future to use the new function manager call interface.

## Complex Queries

A large number of complex queries that were unsupported in previous releases now work. Many combinations of views, aggregates, UNION, LIMIT, cursors, subqueries, and inherited tables now work properly. Inherited tables are now accessed by default. Subqueries in FROM are now supported.

## E.28.1. Migration to version 7.1

A dump/restore using `pg_dump` is required for those wishing to migrate data from any previous release.

## E.28.2. Modificações

### Bug Fixes

```

-----
Many multibyte/Unicode/locale fixes (Tatsuo and others)
More reliable ALTER TABLE RENAME (Tom)
Kerberos V fixes (David Wragg)
Fix for INSERT INTO...SELECT where targetlist has subqueries (Tom)
Prompt username/password on standard error (Bruce)
Large objects inv_read/inv_write fixes (Tom)
Fixes for to_char(), to_date(), to_ascii(), and to_timestamp() (Karel,
    Daniel Baldoni)
Prevent query expressions from leaking memory (Tom)
Allow UPDATE of arrays elements (Tom)
Wake up lock waiters during cancel (Hiroshi)
Fix rare cursor crash when using hash join (Tom)
Fix for DROP TABLE/INDEX in rolled-back transaction (Hiroshi)
Fix psql crash from \l+ if MULTIBYTE enabled (Peter E)
Fix truncation of rule names during CREATE VIEW (Ross Reedstrom)
Fix PL/perl (Alex Kapranoff)
Disallow LOCK on views (Mark Hollomon)
Disallow INSERT/UPDATE/DELETE on views (Mark Hollomon)
Disallow DROP RULE, CREATE INDEX, TRUNCATE on views (Mark Hollomon)
Allow PL/pgSQL accept non-ASCII identifiers (Tatsuo)
Allow views to proper handle GROUP BY, aggregates, DISTINCT (Tom)
Fix rare failure with TRUNCATE command (Tom)
Allow UNION/INTERSECT/EXCEPT to be used with ALL, subqueries, views,
    DISTINCT, ORDER BY, SELECT...INTO (Tom)
Fix parser failures during aborted transactions (Tom)
Allow temporary relations to properly clean up indexes (Bruce)
Fix VACUUM problem with moving rows in same page (Tom)
Modify pg_dump to better handle user-defined items in template1 (Philip)
Allow LIMIT in VIEW (Tom)
Require cursor FETCH to honor LIMIT (Tom)

```

Allow PRIMARY/FOREIGN Key definitions on inherited columns (Stephan)  
 Allow ORDER BY, LIMIT in subqueries (Tom)  
 Allow UNION in CREATE RULE (Tom)  
 Make ALTER/DROP TABLE rollback-able (Vadim, Tom)  
 Store initdb collation in pg\_control so collation cannot be changed (Tom)  
 Fix INSERT...SELECT with rules (Tom)  
 Fix FOR UPDATE inside views and subselects (Tom)  
 Fix OVERLAPS operators conform to SQL92 spec regarding NULLs (Tom)  
 Fix lpad() and rpad() to handle length less than input string (Tom)  
 Fix use of NOTIFY in some rules (Tom)  
 Overhaul btree code (Tom)  
 Fix NOT NULL use in Pl/pgSQL variables (Tom)  
 Overhaul GIST code (Oleg)  
 Fix CLUSTER to preserve constraints and column default (Tom)  
 Improved deadlock detection handling (Tom)  
 Allow multiple SERIAL columns in a table (Tom)  
 Prevent occasional index corruption (Vadim)

#### Enhancements

-----

Add OUTER JOINS (Tom)  
 Function manager overhaul (Tom)  
 Allow ALTER TABLE RENAME on indexes (Tom)  
 Improve CLUSTER (Tom)  
 Improve ps status display for more platforms (Peter E, Marc)  
 Improve CREATE FUNCTION failure message (Ross)  
 JDBC improvements (Peter, Travis Bauer, Christopher Cain, William Webber, Gunnar)  
 Grand Unified Configuration scheme/GUC. Many options can now be set in data/postgresql.conf, postmaster/postgres flags, or SET commands (Peter E)  
 Improved handling of file descriptor cache (Tom)  
 New warning code about auto-created table alias entries (Bruce)  
 Overhaul initdb process (Tom, Peter E)  
 Overhaul of inherited tables; inherited tables now accessed by default; new ONLY key word prevents it (Chris Bitmead, Tom)  
 ODBC cleanups/improvements (Nick Gorham, Stephan Szabo, Zoltan Kovacs, Michael Fork)  
 Allow renaming of temp tables (Tom)  
 Overhaul memory manager contexts (Tom)  
 pg\_dumpall uses CREATE USER or CREATE GROUP rather using COPY (Peter E)  
 Overhaul pg\_dump (Philip Warner)  
 Allow pg\_hba.conf secondary password file to specify only username (Peter E)  
 Allow TEMPORARY or TEMP key word when creating temporary tables (Bruce)  
 New memory leak checker (Karel)  
 New SET SESSION CHARACTERISTICS (Thomas)  
 Allow nested block comments (Thomas)  
 Add WITHOUT TIME ZONE type qualifier (Thomas)  
 New ALTER TABLE ADD CONSTRAINT (Stephan)  
 Use NUMERIC accumulators for INTEGER aggregates (Tom)  
 Overhaul aggregate code (Tom)  
 New VARIANCE and STDDEV() aggregates  
 Improve dependency ordering of pg\_dump (Philip)  
 New pg\_restore command (Philip)  
 New pg\_dump tar output option (Philip)  
 New pg\_dump of large objects (Philip)  
 New ESCAPE option to LIKE (Thomas)  
 New case-insensitive LIKE - ILIKE (Thomas)  
 Allow functional indexes to use binary-compatible type (Tom)  
 Allow SQL functions to be used in more contexts (Tom)  
 New pg\_config utility (Peter E)  
 New PL/pgSQL EXECUTE command which allows dynamic SQL and utility statements (Jan)



New PL/pgSQL GET DIAGNOSTICS statement for SPI value access (Jan)  
 New quote\_identifiers() and quote\_literal() functions (Jan)  
 New ALTER TABLE table OWNER TO user command (Mark Hollomon)  
 Allow subselects in FROM, i.e. FROM (SELECT ...) [AS] alias (Tom)  
 Update PyGreSQL to version 3.1 (D'Arcy)  
 Store tables as files named by OID (Vadim)  
 New SQL function setval(seq,val,bool) for use in pg\_dump (Philip)  
 Require DROP VIEW to remove views, no DROP TABLE (Mark)  
 Allow DROP VIEW view1, view2 (Mark)  
 Allow multiple objects in DROP INDEX, DROP RULE, and DROP TYPE (Tom)  
 Allow automatic conversion to/from Unicode (Tatsuo, Eiji)  
 New /contrib/pgcrypto hashing functions (Marko Kreen)  
 New pg\_dumpall --globals-only option (Peter E)  
 New CHECKPOINT command for WAL which creates new WAL log file (Vadim)  
 New AT TIME ZONE syntax (Thomas)  
 Allow location of Unix domain socket to be configurable (David J. MacKenzie)  
 Allow postmaster to listen on a specific IP address (David J. MacKenzie)  
 Allow socket path name to be specified in hostname by using leading slash  
 (David J. MacKenzie)  
 Allow CREATE DATABASE to specify template database (Tom)  
 New utility to convert MySQL schema dumps to SQL92 and PostgreSQL (Thomas)  
 New /contrib/rserv replication toolkit (Vadim)  
 New file format for COPY BINARY (Tom)  
 New /contrib/oid2name to map numeric files to table names (B Palmer)  
 New "idle in transaction" ps status message (Marc)  
 Update to pgaccess 0.98.7 (Constantin Teodorescu)  
 pg\_ctl now defaults to -w (wait) on shutdown, new -l (log) option  
 Add rudimentary dependency checking to pg\_dump (Philip)

#### Types

-----  
 Fix INET/CIDR type ordering and add new functions (Tom)  
 Make OID behave as an unsigned type (Tom)  
 Allow BIGINT as synonym for INT8 (Peter E)  
 New int2 and int8 comparison operators (Tom)  
 New BIT and BIT VARYING types (Adriaan Joubert, Tom, Peter E)  
 CHAR() no longer faster than VARCHAR() because of TOAST (Tom)  
 New GIST seg/cube examples (Gene Selkov)  
 Improved round(numeric) handling (Tom)  
 Fix CIDR output formatting (Tom)  
 New CIDR abbrev() function (Tom)

#### Performance

-----  
 Write-Ahead Log (WAL) to provide crash recovery with less performance  
 overhead (Vadim)  
 ANALYZE stage of VACUUM no longer exclusively locks table (Bruce)  
 Reduced file seeks (Denis Perchine)  
 Improve BTREE code for duplicate keys (Tom)  
 Store all large objects in a single table (Denis Perchine, Tom)  
 Improve memory allocation performance (Karel, Tom)

#### Source Code

-----  
 New function manager call conventions (Tom)  
 SGI portability fixes (David Kaelbling)  
 New configure --enable-syslog option (Peter E)  
 New BSDI README (Bruce)  
 configure script moved to top level, not /src (Peter E)  
 Makefile/configuration/compilation overhaul (Peter E)  
 New configure --with-python option (Peter E)  
 Solaris cleanups (Peter E)

Overhaul /contrib Makefiles (Karel)  
 New OpenSSL configuration option (Magnus, Peter E)  
 AIX fixes (Andreas)  
 QNX fixes (Maurizio)  
 New heap\_open(), heap\_openr() API (Tom)  
 Remove colon and semi-colon operators (Thomas)  
 New pg\_class.relkind value for views (Mark Hollomon)  
 Rename ichar() to chr() (Karel)  
 New documentation for btrim(), ascii(), chr(), repeat() (Karel)  
 Fixes for NT/Cygwin (Pete Forman)  
 AIX port fixes (Andreas)  
 New BeOS port (David Reid, Cyril Velter)  
 Add proofreader's changes to docs (Addison-Wesley, Bruce)  
 New Alpha spinlock code (Adriaan Joubert, Compaq)  
 UnixWare port overhaul (Peter E)  
 New Darwin/MacOS X port (Peter Bierman, Bruce Hartzler)  
 New FreeBSD Alpha port (Alfred)  
 Overhaul shared memory segments (Tom)  
 Add IBM S/390 support (Neale Ferguson)  
 Moved macmanuf to /contrib (Larry Rosenman)  
 Syslog improvements (Larry Rosenman)  
 New template0 database that contains no user additions (Tom)  
 New /contrib/cube and /contrib/seg GIST sample code (Gene Selkov)  
 Allow NetBSD's libedit instead of readline (Peter)  
 Improved assembly language source code format (Bruce)  
 New contrib/pg\_logger  
 New --template option to createdb  
 New contrib/pg\_control utility (Oliver)  
 New FreeBSD tools ipc\_check, start-scripts/freebsd

## E.29. Release 7.0.3

**Release date:** 2000-11-11

This has a variety of fixes from 7.0.2.

### E.29.1. Migration to version 7.0.3

A dump/restore is *not* required for those running 7.0.\*.

### E.29.2. Modificações

Jdbc fixes (Peter)  
 Large object fix (Tom)  
 Fix lean in COPY WITH OIDS leak (Tom)  
 Fix backwards-index-scan (Tom)  
 Fix SELECT ... FOR UPDATE so it checks for duplicate keys (Hiroshi)  
 Add --enable-syslog to configure (Marc)  
 Fix abort transaction at backend exit in rare cases (Tom)  
 Fix for psql \l+ when multibyte enabled (Tatsuo)  
 Allow PL/pgSQL to accept non ascii identifiers (Tatsuo)  
 Make vacuum always flush buffers (Tom)  
 Fix to allow cancel while waiting for a lock (Hiroshi)  
 Fix for memory allocation problem in user authentication code (Tom)  
 Remove bogus use of int4out() (Tom)  
 Fixes for multiple subqueries in COALESCE or BETWEEN (Tom)  
 Fix for failure of triggers on heap open in certain cases (Jeroen van Vianen)

Fix for erroneous selectivity of not-equals (Tom)  
 Fix for erroneous use of strcmp() (Tom)  
 Fix for bug where storage manager accesses items beyond end of file (Tom)  
 Fix to include kernel errno message in all smgr elog messages (Tom)  
 Fix for '.' not in PATH at build time (SL Baur)  
 Fix for out-of-file-descriptors error (Tom)  
 Fix to make pg\_dump dump 'iscachable' flag for functions (Tom)  
 Fix for subselect in targetlist of Append node (Tom)  
 Fix for mergejoin plans (Tom)  
 Fix TRUNCATE failure on relations with indexes (Tom)  
 Avoid database-wide restart on write error (Hiroshi)  
 Fix nodeMaterial to honor chgParam by recomputing its output (Tom)  
 Fix VACUUM problem with moving chain of update row versions when source and destination of a row version lie on the same page (Tom)  
 Fix user.c CommandCounterIncrement (Tom)  
 Fix for AM/PM boundary problem in to\_char() (Karel Zak)  
 Fix TIME aggregate handling (Tom)  
 Fix to\_char() to avoid coredump on NULL input (Tom)  
 Buffer fix (Tom)  
 Fix for inserting/copying longer multibyte strings into char() data types (Tatsuo)  
 Fix for crash of backend, on abort (Tom)

## E.30. Release 7.0.2

**Release date:** 2000-06-05

This is a repackaging of 7.0.1 with added documentation.

### E.30.1. Migration to version 7.0.2

A dump/restore is *not* required for those running 7.\*.

### E.30.2. Modificações

Added documentation to tarball.

## E.31. Release 7.0.1

**Release date:** 2000-06-01

This is a cleanup release for 7.0.

### E.31.1. Migration to version 7.0.1

A dump/restore is *not* required for those running 7.0.

### E.31.2. Modificações

Fix many CLUSTER failures (Tom)  
 Allow ALTER TABLE RENAME works on indexes (Tom)

Fix plpgsql to handle datetime->timestamp and timespan->interval (Bruce)  
 New configure --with-setproctitle switch to use setproctitle() (Marc, Bruce)  
 Fix the off by one errors in ResultSet from 6.5.3, and more.  
 jdbc ResultSet fixes (Joseph Shraibman)  
 optimizer tunings (Tom)  
 Fix create user for pgaccess  
 Fix for UNLISTEN failure  
 IRIX fixes (David Kaelbling)  
 QNX fixes (Andreas Kardos)  
 Reduce COPY IN lock level (Tom)  
 Change libpqeasy to use PQconnectdb() style parameters (Bruce)  
 Fix pg\_dump to handle OID indexes (Tom)  
 Fix small memory leak (Tom)  
 Solaris fix for createdb/dropdb (Tatsuo)  
 Fix for non-blocking connections (Alfred Perlstein)  
 Fix improper recovery after RENAME TABLE failures (Tom)  
 Copy pg\_ident.conf.sample into /lib directory in install (Bruce)  
 Add SJIS UDC (NEC selection IBM kanji) support (Eiji Tokuya)  
 Fix too long syslog message (Tatsuo)  
 Fix problem with quoted indexes that are too long (Tom)  
 JDBC ResultSet.getTimestamp() fix (Gregory Krasnow & Floyd Marinescu)  
 ecpg changes (Michael)

## E.32. Release 7.0

**Release date:** 2000-05-08

This release contains improvements in many areas, demonstrating the continued growth of PostgreSQL. There are more improvements and fixes in 7.0 than in any previous release. The developers have confidence that this is the best release yet; we do our best to put out only solid releases, and this one is no exception.

Major changes in this release:

### Foreign Keys

Foreign keys are now implemented, with the exception of PARTIAL MATCH foreign keys. Many users have been asking for this feature, and we are pleased to offer it.

### Optimizer Overhaul

Continuing on work started a year ago, the optimizer has been improved, allowing better query plan selection and faster performance with less memory usage.

### Updated psql

psql, our interactive terminal monitor, has been updated with a variety of new features. See the psql manual page for details.

### Join Syntax

SQL92 join syntax is now supported, though only as INNER JOIN for this release. JOIN, NATURAL JOIN, JOIN/USING, and JOIN/ON are available, as are column correlation names.

### E.32.1. Migration to version 7.0

A dump/restore using pg\_dump is required for those wishing to migrate data from any previous release of PostgreSQL. For those upgrading from 6.5.\*, you may instead use pg\_upgrade to upgrade to this release; however, a full dump/reload installation is always the most robust method for upgrades.

Interface and compatibility issues to consider for the new release include:

- The date/time types `datetime` and `timespan` have been superseded by the SQL92-defined types `timestamp` and `interval`. Although there has been some effort to ease the transition by allowing PostgreSQL to recognize the deprecated type names and translate them to the new type names, this mechanism may not be completely transparent to your existing application.
- The optimizer has been substantially improved in the area of query cost estimation. In some cases, this will result in decreased query times as the optimizer makes a better choice for the preferred plan. However, in a small number of cases, usually involving pathological distributions of data, your query times may go up. If you are dealing with large amounts of data, you may want to check your queries to verify performance.
- The JDBC and ODBC interfaces have been upgraded and extended.
- The string function `CHAR_LENGTH` is now a native function. Previous versions translated this into a call to `LENGTH`, which could result in ambiguity with other types implementing `LENGTH` such as the geometric types.

## E.32.2. Modificações

### Bug Fixes

-----

Prevent function calls exceeding maximum number of arguments (Tom)  
 Improve CASE construct (Tom)  
 Fix SELECT coalesce(f1,0) FROM int4\_tbl GROUP BY f1 (Tom)  
 Fix SELECT sentence.words[0] FROM sentence GROUP BY sentence.words[0] (Tom)  
 Fix GROUP BY scan bug (Tom)  
 Improvements in SQL grammar processing (Tom)  
 Fix for views involved in INSERT ... SELECT ... (Tom)  
 Fix for SELECT a/2, a/2 FROM test\_missing\_target GROUP BY a/2 (Tom)  
 Fix for subselects in INSERT ... SELECT (Tom)  
 Prevent INSERT ... SELECT ... ORDER BY (Tom)  
 Fixes for relations greater than 2GB, including vacuum  
 Improve propagating system table changes to other backends (Tom)  
 Improve propagating user table changes to other backends (Tom)  
 Fix handling of temp tables in complex situations (Bruce, Tom)  
 Allow table locking at table open, improving concurrent reliability (Tom)  
 Properly quote sequence names in pg\_dump (Ross J. Reedstrom)  
 Prevent DROP DATABASE while others accessing  
 Prevent any rows from being returned by GROUP BY if no rows processed (Tom)  
 Fix SELECT COUNT(1) FROM table WHERE ... if no rows matching WHERE (Tom)  
 Fix pg\_upgrade so it works for MVCC (Tom)  
 Fix for SELECT ... WHERE x IN (SELECT ... HAVING SUM(x) > 1) (Tom)  
 Fix for "f1 datetime DEFAULT 'now'" (Tom)  
 Fix problems with CURRENT\_DATE used in DEFAULT (Tom)  
 Allow comment-only lines, and ;;; lines too. (Tom)  
 Improve recovery after failed disk writes, disk full (Hiroshi)  
 Fix cases where table is mentioned in FROM but not joined (Tom)  
 Allow HAVING clause without aggregate functions (Tom)  
 Fix for "--" comment and no trailing newline, as seen in perl interface  
 Improve pg\_dump failure error reports (Bruce)  
 Allow sorts and hashes to exceed 2GB file sizes (Tom)  
 Fix for pg\_dump dumping of inherited rules (Tom)  
 Fix for NULL handling comparisons (Tom)  
 Fix inconsistent state caused by failed CREATE/DROP commands (Hiroshi)  
 Fix for dbname with dash  
 Prevent DROP INDEX from interfering with other backends (Tom)  
 Fix file descriptor leak in verify\_password()  
 Fix for "Unable to identify an operator = \$" problem  
 Fix ODBC so no segfault if CommLog and Debug enabled (Dirk Niggemann)  
 Fix for recursive exit call (Massimo)  
 Fix for extra-long timezones (Jeroen van Vianen)

Make pg\_dump preserve primary key information (Peter E)  
 Prevent databases with single quotes (Peter E)  
 Prevent DROP DATABASE inside transaction (Peter E)  
 ecpg memory leak fixes (Stephen Birch)  
 Fix for SELECT null::text, SELECT int4fac(null) and SELECT 2 + (null) (Tom)  
 Y2K timestamp fix (Massimo)  
 Fix for VACUUM 'HEAP\_MOVED\_IN was not expected' errors (Tom)  
 Fix for views with tables/columns containing spaces (Tom)  
 Prevent privileges on indexes (Peter E)  
 Fix for spinlock stuck problem when error is generated (Hiroshi)  
 Fix ipcclean on Linux  
 Fix handling of NULL constraint conditions (Tom)  
 Fix memory leak in odbc driver (Nick Gorham)  
 Fix for privilege check on UNION tables (Tom)  
 Fix to allow SELECT 'a' LIKE 'a' (Tom)  
 Fix for SELECT 1 + NULL (Tom)  
 Fixes to CHAR  
 Fix log() on numeric type (Tom)  
 Deprecate ':' and ';' operators  
 Allow vacuum of temporary tables  
 Disallow inherited columns with the same name as new columns  
 Recover or force failure when disk space is exhausted (Hiroshi)  
 Fix INSERT INTO ... SELECT with AS columns matching result columns  
 Fix INSERT ... SELECT ... GROUP BY groups by target columns not source columns (Tom)  
 Fix CREATE TABLE test (a char(5) DEFAULT text '', b int4)with INSERT (Tom)  
 Fix UNION with LIMIT  
 Fix CREATE TABLE x AS SELECT 1 UNION SELECT 2  
 Fix CREATE TABLE test(col char(2) DEFAULT user)  
 Fix mismatched types in CREATE TABLE ... DEFAULT  
 Fix SELECT \* FROM pg\_class where oid in (0,-1)  
 Fix SELECT COUNT('asdf') FROM pg\_class WHERE oid=12  
 Prevent user who can create databases can modifying pg\_database table (Peter E)  
 Fix btree to give a useful elog when key > 1/2 (page - overhead) (Tom)  
 Fix INSERT of 0.0 into DECIMAL(4,4) field (Tom)

#### Enhancements

-----

New CLI interface include file sqlcli.h, based on SQL3/SQL98  
 Remove all limits on query length, row length limit still exists (Tom)  
 Update jdbc protocol to 2.0 (Jens Glaser <jens@jens.de>)  
 Add TRUNCATE command to quickly truncate relation (Mike Mascari)  
 Fix to give super user and createdb user proper update catalog rights (Peter E)  
 Allow ecpg bool variables to have NULL values (Christof)  
 Issue ecpg error if NULL value for variable with no NULL indicator (Christof)  
 Allow ^C to cancel COPY command (Massimo)  
 Add SET FSYNC and SHOW PG\_OPTIONS commands (Massimo)  
 Function name overloading for dynamically-loaded C functions (Frankpitt)  
 Add CmdTuples() to libpq++ (Vince)  
 New CREATE CONSTRAINT TRIGGER and SET CONSTRAINTS commands (Jan)  
 Allow CREATE FUNCTION/WITH clause to be used for all language types  
 configure --enable-debug adds -g (Peter E)  
 configure --disable-debug removes -g (Peter E)  
 Allow more complex default expressions (Tom)  
 First real FOREIGN KEY constraint trigger functionality (Jan)  
 Add FOREIGN KEY ... MATCH FULL ... ON DELETE CASCADE (Jan)  
 Add FOREIGN KEY ... MATCH <unspecified> referential actions (Don Baccus)  
 Allow WHERE restriction on ctid (physical heap location) (Hiroshi)  
 Move pginterface from contrib to interface directory, rename to pgeasy (Bruce)  
 Change pgeasy connectdb() parameter ordering (Bruce)  
 Require SELECT DISTINCT target list to have all ORDER BY columns (Tom)  
 Add Oracle's COMMENT ON command (Mike Mascari <mascari@yahoo.com>)  
 libpq's PQsetNoticeProcessor function now returns previous hook (Peter E)

Prevent PQsetNoticeProcessor from being set to NULL (Peter E)  
 Make USING in COPY optional (Bruce)  
 Allow subselects in the target list (Tom)  
 Allow subselects on the left side of comparison operators (Tom)  
 New parallel regression test (Jan)  
 Change backend-side COPY to write files with permissions 644 not 666 (Tom)  
 Force permissions on PGDATA directory to be secure, even if it exists (Tom)  
 Added psql LASTOID variable to return last inserted oid (Peter E)  
 Allow concurrent vacuum and remove pg\_vlock vacuum lock file (Tom)  
 Add privilege check for vacuum (Peter E)  
 New libpq functions to allow asynchronous connections: PQconnectStart(),  
     PQconnectPoll(), PQresetStart(), PQresetPoll(), PQsetenvStart(),  
     PQsetenvPoll(), PQsetenvAbort (Ewan Mellor)  
 New libpq PQsetenv() function (Ewan Mellor)  
 create/alter user extension (Peter E)  
 New postmaster.pid and postmaster.opts under \$PGDATA (Tatsuo)  
 New scripts for create/drop user/db (Peter E)  
 Major psql overhaul (Peter E)  
 Add const to libpq interface (Peter E)  
 New libpq function PQoidValue (Peter E)  
 Show specific non-aggregate causing problem with GROUP BY (Tom)  
 Make changes to pg\_shadow recreate pg\_pwd file (Peter E)  
 Add aggregate(DISTINCT ...) (Tom)  
 Allow flag to control COPY input/output of NULLs (Peter E)  
 Make postgres user have a password by default (Peter E)  
 Add CREATE/ALTER/DROP GROUP (Peter E)  
 All administration scripts now support --long options (Peter E, Karel)  
 Vacuumdb script now supports --all option (Peter E)  
 ecpg new portable FETCH syntax  
 Add ecpg EXEC SQL IFDEF, EXEC SQL IFNDEF, EXEC SQL ELSE, EXEC SQL ELIF  
     and EXEC SQL ENDIF directives  
 Add pg\_ctl script to control backend start-up (Tatsuo)  
 Add postmaster.opts.default file to store start-up flags (Tatsuo)  
 Allow --with-mb=SQL\_ASCII  
 Increase maximum number of index keys to 16 (Bruce)  
 Increase maximum number of function arguments to 16 (Bruce)  
 Allow configuration of maximum number of index keys and arguments (Bruce)  
 Allow unprivileged users to change their passwords (Peter E)  
 Password authentication enabled; required for new users (Peter E)  
 Disallow dropping a user who owns a database (Peter E)  
 Change initdb option --with-mb to --enable-multibyte  
 Add option for initdb to prompts for superuser password (Peter E)  
 Allow complex type casts like col::numeric(9,2) and col::int2::float8 (Tom)  
 Updated user interfaces on initdb, initlocation, pg\_dump, ipcclean (Peter E)  
 New pg\_char\_to\_encoding() and pg\_encoding\_to\_char() functions (Tatsuo)  
 libpq non-blocking mode (Alfred Perlstein)  
 Improve conversion of types in casts that don't specify a length  
 New plperl internal programming language (Mark Hollomon)  
 Allow COPY IN to read file that do not end with a newline (Tom)  
 Indicate when long identifiers are truncated (Tom)  
 Allow aggregates to use type equivalency (Peter E)  
 Add Oracle's to\_char(), to\_date(), to\_datetime(), to\_timestamp(), to\_number()  
     conversion functions (Karel Zak <zakkr@zf.jcu.cz>)  
 Add SELECT DISTINCT ON (expr [, expr ...]) targetlist ... (Tom)  
 Check to be sure ORDER BY is compatible with the DISTINCT operation (Tom)  
 Add NUMERIC and int8 types to ODBC  
 Improve EXPLAIN results for Append, Group, Agg, Unique (Tom)  
 Add ALTER TABLE ... ADD FOREIGN KEY (Stephan Szabo)  
 Allow SELECT .. FOR UPDATE in PL/pgSQL (Hiroshi)  
 Enable backward sequential scan even after reaching EOF (Hiroshi)  
 Add btree indexing of boolean values, >= and <= (Don Baccus)  
 Print current line number when COPY FROM fails (Massimo)

Recognize POSIX time zone e.g. "PST+8" and "GMT-8" (Thomas)  
 Add DEC as synonym for DECIMAL (Thomas)  
 Add SESSION\_USER as SQL92 key word, same as CURRENT\_USER (Thomas)  
 Implement SQL92 column aliases (aka correlation names) (Thomas)  
 Implement SQL92 join syntax (Thomas)  
 Make INTERVAL reserved word allowed as a column identifier (Thomas)  
 Implement REINDEX command (Hiroshi)  
 Accept ALL in aggregate function SUM(ALL col) (Tom)  
 Prevent GROUP BY from using column aliases (Tom)  
 New psql \encoding option (Tatsuo)  
 Allow PQrequestCancel() to terminate when in waiting-for-lock state (Hiroshi)  
 Allow negation of a negative number in all cases  
 Add ecpg descriptors (Christof, Michael)  
 Allow CREATE VIEW v AS SELECT fl::char(8) FROM tbl  
 Allow casts with length, like foo::char(8)  
 New libpq functions PQsetClientEncoding(), PQclientEncoding() (Tatsuo)  
 Add support for SJIS user defined characters (Tatsuo)  
 Larger views/rules supported  
 Make libpq's PQconndefaults() thread-safe (Tom)  
 Disable // as comment to be ANSI conforming, should use -- (Tom)  
 Allow column aliases on views CREATE VIEW name (collist)  
 Fixes for views with subqueries (Tom)  
 Allow UPDATE table SET fld = (SELECT ...) (Tom)  
 SET command options no longer require quotes  
 Update pgaccess to 0.98.6  
 New SET SEED command  
 New pg\_options.sample file  
 New SET FSYNC command (Massimo)  
 Allow pg\_descriptions when creating tables  
 Allow pg\_descriptions when creating types, columns, and functions  
 Allow psql \copy to allow delimiters (Peter E)  
 Allow psql to print nulls as distinct from "" [null] (Peter E)

## Types

-----

Many array fixes (Tom)  
 Allow bare column names to be subscripted as arrays (Tom)  
 Improve type casting of int and float constants (Tom)  
 Cleanups for int8 inputs, range checking, and type conversion (Tom)  
 Fix for SELECT timespan('21:11:26'::time) (Tom)  
 netmask('x.x.x.x/0') is 255.255.255.255 instead of 0.0.0.0 (Oleg Sharoiko)  
 Add btree index on NUMERIC (Jan)  
 Perl fix for large objects containing NUL characters (Douglas Thomson)  
 ODBC fix for for large objects (free)  
 Fix indexing of cidr data type  
 Fix for Ethernet MAC addresses (macaddr type) comparisons  
 Fix for date/time types when overflows happened in computations (Tom)  
 Allow array on int8 (Peter E)  
 Fix for rounding/overflow of NUMERIC type, like NUMERIC(4,4) (Tom)  
 Allow NUMERIC arrays  
 Fix bugs in NUMERIC ceil() and floor() functions (Tom)  
 Make char\_length()/octet\_length including trailing blanks (Tom)  
 Made abstime/retime use int4 instead of time\_t (Peter E)  
 New lztext data type for compressed text fields  
 Revise code to handle coercion of int and float constants (Tom)  
 Start at new code to implement a BIT and BIT VARYING type (Adriaan Joubert)  
 NUMERIC now accepts scientific notation (Tom)  
 NUMERIC to int4 rounds (Tom)  
 Convert float4/8 to NUMERIC properly (Tom)  
 Allow type conversion with NUMERIC (Thomas)  
 Make ISO date style (2000-02-16 09:33) the default (Thomas)  
 Add NATIONAL CHAR [ VARYING ] (Thomas)



Allow NUMERIC round and trunc to accept negative scales (Tom)  
 New TIME WITH TIME ZONE type (Thomas)  
 Add MAX()/MIN() on time type (Thomas)  
 Add abs(), mod(), fac() for int8 (Thomas)  
 Rename functions to round(), sqrt(), cbrt(), pow() for float8 (Thomas)  
 Add transcendental math functions (e.g. sin(), acos()) for float8 (Thomas)  
 Add exp() and ln() for NUMERIC type  
 Rename NUMERIC power() to pow() (Thomas)  
 Improved TRANSLATE() function (Edwin Ramirez, Tom)  
 Allow X=-Y operators (Tom)  
 Allow SELECT float8(COUNT(\*))/(SELECT COUNT(\*) FROM t) FROM t GROUP BY f1; (Tom)  
 Allow LOCALE to use indexes in regular expression searches (Tom)  
 Allow creation of functional indexes to use default types

## Performance

-----  
 Prevent exponential space consumption with many AND's and OR's (Tom)  
 Collect attribute selectivity values for system columns (Tom)  
 Reduce memory usage of aggregates (Tom)  
 Fix for LIKE optimization to use indexes with multibyte encodings (Tom)  
 Fix r-tree index optimizer selectivity (Thomas)  
 Improve optimizer selectivity computations and functions (Tom)  
 Optimize btree searching for cases where many equal keys exist (Tom)  
 Enable fast LIKE index processing only if index present (Tom)  
 Re-use free space on index pages with duplicates (Tom)  
 Improve hash join processing (Tom)  
 Prevent descending sort if result is already sorted (Hiroshi)  
 Allow commuting of index scan query qualifications (Tom)  
 Prefer index scans in cases where ORDER BY/GROUP BY is required (Tom)  
 Allocate large memory requests in fix-sized chunks for performance (Tom)  
 Fix vacuum's performance by reducing memory allocation requests (Tom)  
 Implement constant-expression simplification (Bernard Frankpitt, Tom)  
 Use secondary columns to be used to determine start of index scan (Hiroshi)  
 Prevent quadruple use of disk space when doing internal sorting (Tom)  
 Faster sorting by calling fewer functions (Tom)  
 Create system indexes to match all system caches (Bruce, Hiroshi)  
 Make system caches use system indexes (Bruce)  
 Make all system indexes unique (Bruce)  
 Improve pg\_statistics management for VACUUM speed improvement (Tom)  
 Flush backend cache less frequently (Tom, Hiroshi)  
 COPY now reuses previous memory allocation, improving performance (Tom)  
 Improve optimization cost estimation (Tom)  
 Improve optimizer estimate of range queries  $x > \text{lowbound}$  AND  $x < \text{highbound}$  (Tom)  
 Use DNF instead of CNF where appropriate (Tom, Taral)  
 Further cleanup for OR-of-AND WHERE-clauses (Tom)  
 Make use of index in OR clauses ( $x = 1$  AND  $y = 2$ ) OR ( $x = 2$  AND  $y = 4$ ) (Tom)  
 Smarter optimizer computations for random index page access (Tom)  
 New SET variable to control optimizer costs (Tom)  
 Optimizer queries based on LIMIT, OFFSET, and EXISTS qualifications (Tom)  
 Reduce optimizer internal housekeeping of join paths for speedup (Tom)  
 Major subquery speedup (Tom)  
 Fewer fsync writes when fsync is not disabled (Tom)  
 Improved LIKE optimizer estimates (Tom)  
 Prevent fsync in SELECT-only queries (Vadim)  
 Make index creation use psort code, because it is now faster (Tom)  
 Allow creation of sort temp tables > 1 Gig

## Source Tree Changes

-----  
 Fix for linux PPC compile  
 New generic expression-tree-walker subroutine (Tom)  
 Change form() to varargform() to prevent portability problems

Improved range checking for large integers on Alphas  
 Clean up #include in /include directory (Bruce)  
 Add scripts for checking includes (Bruce)  
 Remove un-needed #include's from \*.c files (Bruce)  
 Change #include's to use <> and "" as appropriate (Bruce)  
 Enable Windows compilation of libpq  
 Alpha spinlock fix from Uncle George <gatgul@voicenet.com>  
 Overhaul of optimizer data structures (Tom)  
 Fix to cygipc library (Yutaka Tanida)  
 Allow pgsqll to work on newer Cygwin snapshots (Dan)  
 New catalog version number (Tom)  
 Add Linux ARM  
 Rename heap\_replace to heap\_update  
 Update for QNX (Dr. Andreas Kardos)  
 New platform-specific regression handling (Tom)  
 Rename oid8 -> oidvector and int28 -> int2vector (Bruce)  
 Included all yacc and lex files into the distribution (Peter E.)  
 Remove lexttest, no longer needed (Peter E.)  
 Fix for libpq and psql on Windows (Magnus)  
 Internally change datetime and timespan into timestamp and interval (Thomas)  
 Fix for plpgsql on BSD/OS  
 Add SQL\_ASCII test case to the regression test (Tatsuo)  
 configure --with-mb now deprecated (Tatsuo)  
 NT fixes  
 NetBSD fixes (Johnny C. Lam <lamj@stat.cmu.edu>)  
 Fixes for Alpha compiles  
 New multibyte encodings

## E.33. Release 6.5.3

**Release date:** 1999-10-13

This is basically a cleanup release for 6.5.2. We have added a new PgAccess that was missing in 6.5.2, and installed an NT-specific fix.

### E.33.1. Migration to version 6.5.3

A dump/restore is *not* required for those running 6.5.\*.

### E.33.2. Modificações

Updated version of pgaccess 0.98  
 NT-specific patch  
 Fix dumping rules on inherited tables

## E.34. Release 6.5.2

**Release date:** 1999-09-15

This is basically a cleanup release for 6.5.1. We have fixed a variety of problems reported by 6.5.1 users.

### E.34.1. Migration to version 6.5.2

A dump/restore is *not* required for those running 6.5.\*.

## E.34.2. Modificações

subselect+CASE fixes(Tom)  
 Add SHLIB\_LINK setting for solaris\_i386 and solaris\_sparc ports(Daren Sefcik)  
 Fixes for CASE in WHERE join clauses(Tom)  
 Fix BTScan abort(Tom)  
 Repair the check for redundant UNIQUE and PRIMARY KEY indexes(Thomas)  
 Improve it so that it checks for multicolumn constraints(Thomas)  
 Fix for Windows making problem with MB enabled(Hiroki Kataoka)  
 Allow BSD yacc and bison to compile pl code(Bruce)  
 Fix SET NAMES working  
 int8 fixes(Thomas)  
 Fix vacuum's memory consumption(Hiroshi,Tatsuo)  
 Reduce the total memory consumption of vacuum(Tom)  
 Fix for timestamp(datetime)  
 Rule deparsing bugfixes(Tom)  
 Fix quoting problems in mkMakefile.tcldefs.sh.in and mkMakefile.tkdefs.sh.in(Tom)  
 This is to re-use space on index pages freed by vacuum(Vadim)  
 document -x for pg\_dump(Bruce)  
 Fix for unary operators in rule parser(Tom)  
 Comment out FileUnlink of excess segments during mdtruncate()(Tom)  
 IRIX linking fix from Yu Cao >yucaofalcon.kla-tencor.com<  
 Repair logic error in LIKE: should not return LIKE\_ABORT  
     when reach end of pattern before end of text(Tom)  
 Repair incorrect cleanup of heap memory allocation during transaction abort(Tom)  
 Updated version of pgaccess 0.98

## E.35. Release 6.5.1

**Release date:** 1999-07-15

This is basically a cleanup release for 6.5. We have fixed a variety of problems reported by 6.5 users.

### E.35.1. Migration to version 6.5.1

A dump/restore is *not* required for those running 6.5.

## E.35.2. Modificações

Add NT README file  
 Portability fixes for linux\_ppc, IRIX, linux\_alpha, OpenBSD, alpha  
 Remove QUERY\_LIMIT, use SELECT...LIMIT  
 Fix for EXPLAIN on inheritance(Tom)  
 Patch to allow vacuum on multisegment tables(Hiroshi)  
 R-Tree optimizer selectivity fix(Tom)  
 ACL file descriptor leak fix(Atsushi Ogawa)  
 New expression subtree code(Tom)  
 Avoid disk writes for read-only transactions(Vadim)  
 Fix for removal of temp tables if last transaction was aborted(Bruce)  
 Fix to prevent too large row from being created(Bruce)  
 plpgsql fixes  
 Allow port numbers 32k - 64k(Bruce)  
 Add ^ precedence(Bruce)  
 Rename sort files called pg\_temp to pg\_sorttemp(Bruce)  
 Fix for microseconds in time values(Tom)

Tutorial source cleanup  
 New linux\_m68k port  
 Fix for sorting of NULL's in some cases(Tom)  
 Shared library dependencies fixed (Tom)  
 Fixed glitches affecting GROUP BY in subselects(Tom)  
 Fix some compiler warnings (Tomoaki Nishiyama)  
 Add Win1250 (Czech) support (Pavel Behal)

## E.36. Release 6.5

**Release date:** 1999-06-09

This release marks a major step in the development team's mastery of the source code we inherited from Berkeley. You will see we are now easily adding major features, thanks to the increasing size and experience of our world-wide development team.

Here is a brief summary of the more notable changes:

### Multiversion concurrency control(MVCC)

This removes our old table-level locking, and replaces it with a locking system that is superior to most commercial database systems. In a traditional system, each row that is modified is locked until committed, preventing reads by other users. MVCC uses the natural multiversion nature of PostgreSQL to allow readers to continue reading consistent data during writer activity. Writers continue to use the compact pg\_log transaction system. This is all performed without having to allocate a lock for every row like traditional database systems. So, basically, we no longer are restricted by simple table-level locking; we have something better than row-level locking.

### Hot backups from pg\_dump

pg\_dump takes advantage of the new MVCC features to give a consistent database dump/backup while the database stays online and available for queries.

### Numeric data type

We now have a true numeric data type, with user-specified precision.

### Temporary tables

Temporary tables are guaranteed to have unique names within a database session, and are destroyed on session exit.

### New SQL features

We now have CASE, INTERSECT, and EXCEPT statement support. We have new LIMIT/OFFSET, SET TRANSACTION ISOLATION LEVEL, SELECT ... FOR UPDATE, and an improved LOCK TABLE command.

### Speedups

We continue to speed up PostgreSQL, thanks to the variety of talents within our team. We have sped up memory allocation, optimization, table joins, and row transfer routines.

### Ports

We continue to expand our port list, this time including Windows NT/ix86 and NetBSD/arm32.

### Interfaces

Most interfaces have new versions, and existing functionality has been improved.

### Documentation

New and updated material is present throughout the documentation. New FAQs have been contributed for SGI and AIX platforms. The *Tutorial* has introductory information on SQL from Stefan Simkovics. For the *User's Guide*, there are reference pages covering the postmaster and more utility programs, and a new appendix contains details on date/time behavior. The *Administrator's Guide* has a new chapter on troubleshooting from Tom Lane. And the *Programmer's Guide* has a description of query processing, also from Stefan, and details on obtaining the PostgreSQL source tree via anonymous CVS and CVSup.

## E.36.1. Migration to version 6.5

A dump/restore using `pg_dump` is required for those wishing to migrate data from any previous release of PostgreSQL. `pg_upgrade` can *not* be used to upgrade to this release because the on-disk structure of the tables has changed compared to previous releases.

The new Multiversion Concurrency Control (MVCC) features can give somewhat different behaviors in multiuser environments. *Read and understand the following section to ensure that your existing applications will give you the behavior you need.*

### E.36.1.1. Multiversion Concurrency Control

Because readers in 6.5 don't lock data, regardless of transaction isolation level, data read by one transaction can be overwritten by another. In other words, if a row is returned by `SELECT` it doesn't mean that this row really exists at the time it is returned (i.e. sometime after the statement or transaction began) nor that the row is protected from being deleted or updated by concurrent transactions before the current transaction does a commit or rollback.

To ensure the actual existence of a row and protect it against concurrent updates one must use `SELECT FOR UPDATE` or an appropriate `LOCK TABLE` statement. This should be taken into account when porting applications from previous releases of PostgreSQL and other environments.

Keep the above in mind if you are using `contrib/refint.*` triggers for referential integrity. Additional techniques are required now. One way is to use `LOCK parent_table IN SHARE ROW EXCLUSIVE MODE` command if a transaction is going to update/delete a primary key and use `LOCK parent_table IN SHARE MODE` command if a transaction is going to update/insert a foreign key.

**Nota:** Note that if you run a transaction in `SERIALIZABLE` mode then you must execute the `LOCK` commands above before execution of any DML statement (`SELECT/INSERT/DELETE/UPDATE/FETCH/COPY_TO`) in the transaction.

These inconveniences will disappear in the future when the ability to read dirty (uncommitted) data (regardless of isolation level) and true referential integrity will be implemented.

## E.36.2. Modificações

### Bug Fixes

-----

Fix `text<->float8` and `text<->float4` conversion functions(Thomas)  
 Fix for creating tables with mixed-case constraints(Billy)  
 Change `exp()/pow()` behavior to generate error on underflow/overflow(Jan)  
 Fix bug in `pg_dump -z`  
 Memory overrun cleanups(Tatsuo)  
 Fix for `lo_import` crash(Tatsuo)  
 Adjust handling of data type names to suppress double quotes(Thomas)  
 Use type coercion for matching columns and `DEFAULT`(Thomas)  
 Fix deadlock so it only checks once after one second of sleep(Bruce)  
 Fixes for aggregates and PL/pgsql(Hiroshi)  
 Fix for subquery crash(Vadim)  
 Fix for `libpq` function `PQfnumber` and case-insensitive names(Bahman Rafatjoo)  
 Fix for large object write-in-middle, no extra block, memory consumption(Tatsuo)  
 Fix for `pg_dump -d` or `-D` and quote special characters in `INSERT`  
 Repair serious problems with `dynahash`(Tom)  
 Fix `INET/CIDR` portability problems  
 Fix problem with selectivity error in `ALTER TABLE ADD COLUMN`(Bruce)  
 Fix executor so mergejoin of different column types works(Tom)  
 Fix for Alpha OR selectivity bug  
 Fix OR index selectivity problem(Bruce)  
 Fix so `\d` shows proper length for `char()/varchar()`(Ryan)

Fix tutorial code(Clark)  
 Improve destroyuser checking(Oliver)  
 Fix for Kerberos(Rodney McDuff)  
 Fix for dropping database while dirty buffers(Bruce)  
 Fix so sequence nextval() can be case-sensitive(Bruce)  
 Fix != operator  
 Drop buffers before destroying database files(Bruce)  
 Fix case where executor evaluates functions twice(Tatsuo)  
 Allow sequence nextval actions to be case-sensitive(Bruce)  
 Fix optimizer indexing not working for negative numbers(Bruce)  
 Fix for memory leak in executor with fjIsNull  
 Fix for aggregate memory leaks(Erik Riedel)  
 Allow user name containing a dash to grant privileges  
 Cleanup of NULL in inet types  
 Clean up system table bugs(Tom)  
 Fix problems of PAGER and \? command(Masaaki Sakaida)  
 Reduce default multisegment file size limit to 1GB(Peter)  
 Fix for dumping of CREATE OPERATOR(Tom)  
 Fix for backward scanning of cursors(Hiroshi Inoue)  
 Fix for COPY FROM STDIN when using \i(Tom)  
 Fix for subselect is compared inside an expression(Jan)  
 Fix handling of error reporting while returning rows(Tom)  
 Fix problems with reference to array types(Tom,Jan)  
 Prevent UPDATE SET oid(Jan)  
 Fix pg\_dump so -t option can handle case-sensitive tablenames  
 Fixes for GROUP BY in special cases(Tom, Jan)  
 Fix for memory leak in failed queries(Tom)  
 DEFAULT now supports mixed-case identifiers(Tom)  
 Fix for multisegment uses of DROP/RENAME table, indexes(Ole Gjerde)  
 Disable use of pg\_dump with both -o and -d options(Bruce)  
 Allow pg\_dump to properly dump group privileges(Bruce)  
 Fix GROUP BY in INSERT INTO table SELECT \* FROM table2(Jan)  
 Fix for computations in views(Jan)  
 Fix for aggregates on array indexes(Tom)  
 Fix for DEFAULT handles single quotes in value requiring too many quotes  
 Fix security problem with non-super users importing/exporting large objects(Tom)  
 Rollback of transaction that creates table cleaned up properly(Tom)  
 Fix to allow long table and column names to generate proper serial names(Tom)

#### Enhancements

-----  
 Add "vacuumdb" utility  
 Speed up libpq by allocating memory better(Tom)  
 EXPLAIN all indexes used(Tom)  
 Implement CASE, COALESCE, NULLIF expression(Thomas)  
 New pg\_dump table output format(Constantin)  
 Add string min()/max() functions(Thomas)  
 Extend new type coercion techniques to aggregates(Thomas)  
 New moddatetime contrib(Terry)  
 Update to pgaccess 0.96(Constantin)  
 Add routines for single-byte "char" type(Thomas)  
 Improved substr() function(Thomas)  
 Improved multibyte handling(Tatsuo)  
 Multiversion concurrency control/MVCC(Vadim)  
 New Serialized mode(Vadim)  
 Fix for tables over 2gigs(Peter)  
 New SET TRANSACTION ISOLATION LEVEL(Vadim)  
 New LOCK TABLE IN ... MODE(Vadim)  
 Update ODBC driver(Byron)  
 New NUMERIC data type(Jan)  
 New SELECT FOR UPDATE(Vadim)  
 Handle "NaN" and "Infinity" for input values(Jan)

Improved date/year handling(Thomas)  
 Improved handling of backend connections(Magnus)  
 New options ELOG\_TIMESTAMPS and USE\_SYSLOG options for log files(Massimo)  
 New TCL\_ARRAYS option(Massimo)  
 New INTERSECT and EXCEPT(Stefan)  
 New pg\_index.indisprimary for primary key tracking(D'Arcy)  
 New pg\_dump option to allow dropping of tables before creation(Brook)  
 Speedup of row output routines(Tom)  
 New READ COMMITTED isolation level(Vadim)  
 New TEMP tables/indexes(Bruce)  
 Prevent sorting if result is already sorted(Jan)  
 New memory allocation optimization(Jan)  
 Allow psql to do \p\g(Bruce)  
 Allow multiple rule actions(Jan)  
 Added LIMIT/OFFSET functionality(Jan)  
 Improve optimizer when joining a large number of tables(Bruce)  
 New intro to SQL from S. Simkovics' Master's Thesis (Stefan, Thomas)  
 New intro to backend processing from S. Simkovics' Master's Thesis (Stefan)  
 Improved int8 support(Ryan Bradetich, Thomas, Tom)  
 New routines to convert between int8 and text/varchar types(Thomas)  
 New bushy plans, where meta-tables are joined(Bruce)  
 Enable right-hand queries by default(Bruce)  
 Allow reliable maximum number of backends to be set at configure time  
 (--with-maxbackends and postmaster switch (-N backends))(Tom)  
 GEQO default now 10 tables because of optimizer speedups(Tom)  
 Allow NULL=Var for MS-SQL portability(Michael, Bruce)  
 Modify contrib check\_primary\_key() so either "automatic" or "dependent"(Anand)  
 Allow psql \d on a view show query(Ryan)  
 Speedup for LIKE(Bruce)  
 Ecpq fixes/features, see src/interfaces/ecpq/ChangeLog file(Michael)  
 JDBC fixes/features, see src/interfaces/jdbc/CHANGELOG(Peter)  
 Make % operator have precedence like /(Bruce)  
 Add new postgres -O option to allow system table structure changes(Bruce)  
 Update contrib/pginterface/findoidjoins script(Tom)  
 Major speedup in vacuum of deleted rows with indexes(Vadim)  
 Allow non-SQL functions to run different versions based on arguments(Tom)  
 Add -E option that shows actual queries sent by \dt and friends(Masaaki Sakaida)  
 Add version number in start-up banners for psql(Masaaki Sakaida)  
 New contrib/vacuumlo removes large objects not referenced(Peter)  
 New initialization for table sizes so non-vacuumed tables perform better(Tom)  
 Improve error messages when a connection is rejected(Tom)  
 Support for arrays of char() and varchar() fields(Massimo)  
 Overhaul of hash code to increase reliability and performance(Tom)  
 Update to PyGreSQL 2.4(D'Arcy)  
 Changed debug options so -d4 and -d5 produce different node displays(Jan)  
 New pg\_options: pretty\_plan, pretty\_parse, pretty\_rewritten(Jan)  
 Better optimization statistics for system table access(Tom)  
 Better handling of non-default block sizes(Massimo)  
 Improve GEQO optimizer memory consumption(Tom)  
 UNION now supports ORDER BY of columns not in target list(Jan)  
 Major libpq++ improvements(Vince Vielhaber)  
 pg\_dump now uses -z(ACL's) as default(Bruce)  
 backend cache, memory speedups(Tom)  
 have pg\_dump do everything in one snapshot transaction(Vadim)  
 fix for large object memory leakage, fix for pg\_dumping(Tom)  
 INET type now respects netmask for comparisons  
 Make VACUUM ANALYZE only use a readlock(Vadim)  
 Allow VIEWS on UNIONS(Jan)  
 pg\_dump now can generate consistent snapshots on active databases(Vadim)

Source Tree Changes

-----

Improve port matching(Tom)  
 Portability fixes for SunOS  
 Add Windows NT backend port and enable dynamic loading(Magnus and Daniel Horak)  
 New port to Cobalt Qube(Mips) running Linux(Tatsuo)  
 Port to NetBSD/m68k(Mr. Mutsuki Nakajima)  
 Port to NetBSD/sun3(Mr. Mutsuki Nakajima)  
 Port to NetBSD/macppc(Toshimi Aoki)  
 Fix for tcl/tk configuration(Vince)  
 Removed CURRENT key word for rule queries(Jan)  
 NT dynamic loading now works(Daniel Horak)  
 Add ARM32 support(Andrew McMurtry)  
 Better support for HP-UX 11 and UnixWare  
 Improve file handling to be more uniform, prevent file descriptor leak(Tom)  
 New install commands for plpgsql(Jan)

## E.37. Release 6.4.2

**Release date:** 1998-12-20

The 6.4.1 release was improperly packaged. This also has one additional bug fix.

### E.37.1. Migration to version 6.4.2

A dump/restore is *not* required for those running 6.4.\*.

### E.37.2. Modificações

Fix for datetime constant problem on some platforms(Thomas)

## E.38. Release 6.4.1

**Release date:** 1998-12-18

This is basically a cleanup release for 6.4. We have fixed a variety of problems reported by 6.4 users.

### E.38.1. Migration to version 6.4.1

A dump/restore is *not* required for those running 6.4.

### E.38.2. Modificações

Add pg\_dump -N flag to force double quotes around identifiers. This is  
 the default(Thomas)  
 Fix for NOT in where clause causing crash(Bruce)  
 EXPLAIN VERBOSE coredump fix(Vadim)  
 Fix shared-library problems on Linux  
 Fix test for table existence to allow mixed-case and whitespace in  
 the table name(Thomas)  
 Fix a couple of pg\_dump bugs  
 Configure matches template/.similar entries better(Tom)  
 Change builtin function names from SPI\_\* to spi\_\*  
 OR WHERE clause fix(Vadim)  
 Fixes for mixed-case table names(Billy)  
 contrib/linux/postgres.init.csh/sh fix(Thomas)  
 libpq memory overrun fix  
 SunOS fixes(Tom)  
 Change exp() behavior to generate error on underflow(Thomas)  
 pg\_dump fixes for memory leak, inheritance constraints, layout change



update pgaccess to 0.93  
 Fix prototype for 64-bit platforms  
 Multibyte fixes(Tatsuo)  
 New ecpg man page  
 Fix memory overruns(Tatsuo)  
 Fix for lo\_import() crash(Bruce)  
 Better search for install program(Tom)  
 Timezone fixes(Tom)  
 HP-UX fixes(Tom)  
 Use implicit type coercion for matching DEFAULT values(Thomas)  
 Add routines to help with single-byte (internal) character type(Thomas)  
 Compilation of libpq for Windows fixes(Magnus)  
 Upgrade to PyGreSQL 2.2(D'Arcy)

## E.39. Release 6.4

**Release date:** 1998-10-30

There are *many* new features and improvements in this release. Thanks to our developers and maintainers, nearly every aspect of the system has received some attention since the previous release. Here is a brief, incomplete summary:

- Views and rules are now functional thanks to extensive new code in the rewrite rules system from Jan Wieck. He also wrote a chapter on it for the *Programmer's Guide*.
- Jan also contributed a second procedural language, PL/pgSQL, to go with the original PL/pgTCL procedural language he contributed last release.
- We have optional multiple-byte character set support from Tatsuo Ishii to complement our existing locale support.
- Client/server communications has been cleaned up, with better support for asynchronous messages and interrupts thanks to Tom Lane.
- The parser will now perform automatic type coercion to match arguments to available operators and functions, and to match columns and expressions with target columns. This uses a generic mechanism which supports the type extensibility features of PostgreSQL. There is a new chapter in the *User's Guide* which covers this topic.
- Three new data types have been added. Two types, `inet` and `cidr`, support various forms of IP network, subnet, and machine addressing. There is now an 8-byte integer type available on some platforms. See the chapter on data types in the *User's Guide* for details. A fourth type, `serial`, is now supported by the parser as an amalgam of the `int4` type, a sequence, and a unique index.
- Several more SQL92-compatible syntax features have been added, including `INSERT DEFAULT VALUES`
- The automatic configuration and installation system has received some attention, and should be more robust for more platforms than it has ever been.

### E.39.1. Migration to version 6.4

A dump/restore using `pg_dump` or `pg_dumpall` is required for those wishing to migrate data from any previous release of PostgreSQL.

### E.39.2. Modificações

Bug Fixes

-----

Fix for a tiny memory leak in PQsetdb/PQfinish(Bryan)  
 Remove char2-16 data types, use char/varchar(Darren)  
 Pqfn not handles a NOTICE message(Anders)  
 Reduced busywaiting overhead for spinlocks with many backends (dg)  
 Stuck spinlock detection (dg)  
 Fix up "ISO-style" timespan decoding and encoding(Thomas)  
 Fix problem with table drop after rollback of transaction(Vadim)

Change error message and remove non-functional update message(Vadim)  
 Fix for COPY array checking  
 Fix for SELECT 1 UNION SELECT NULL  
 Fix for buffer leaks in large object calls(Pascal)  
 Change owner from oid to int4 type(Bruce)  
 Fix a bug in the oracle compatibility functions btrim() ltrim() and rtrim()  
 Fix for shared invalidation cache overflow(Massimo)  
 Prevent file descriptor leaks in failed COPY's(Bruce)  
 Fix memory leak in libpgtcl's pg\_select(Constantin)  
 Fix problems with username/passwords over 8 characters(Tom)  
 Fix problems with handling of asynchronous NOTIFY in backend(Tom)  
 Fix of many bad system table entries(Tom)

#### Enhancements

-----  
 Upgrade ecpg and ecpglib, see src/interfaces/ecpc/ChangeLog(Michael)  
 Show the index used in an EXPLAIN(Zeugswetter)  
 EXPLAIN invokes rule system and shows plan(s) for rewritten queries(Jan)  
 Multibyte awareness of many data types and functions, via configure(Tatsuo)  
 New configure --with-mb option(Tatsuo)  
 New initdb --pgencoding option(Tatsuo)  
 New createdb -E multibyte option(Tatsuo)  
 Select version(); now returns PostgreSQL version(Jeroen)  
 libpq now allows asynchronous clients(Tom)  
 Allow cancel from client of backend query(Tom)  
 psql now cancels query with Control-C(Tom)  
 libpq users need not issue dummy queries to get NOTIFY messages(Tom)  
 NOTIFY now sends sender's PID, so you can tell whether it was your own(Tom)  
 PGresult struct now includes associated error message, if any(Tom)  
 Define "tz\_hour" and "tz\_minute" arguments to date\_part()(Thomas)  
 Add routines to convert between varchar and bpchar(Thomas)  
 Add routines to allow sizing of varchar and bpchar into target columns(Thomas)  
 Add bit flags to support timezonehour and minute in data retrieval(Thomas)  
 Allow more variations on valid floating point numbers (e.g. ".1", "1e6")(Thomas)  
 Fixes for unary minus parsing with leading spaces(Thomas)  
 Implement TIMEZONE\_HOUR, TIMEZONE\_MINUTE per SQL92 specs(Thomas)  
 Check for and properly ignore FOREIGN KEY column constraints(Thomas)  
 Define USER as synonym for CURRENT\_USER per SQL92 specs(Thomas)  
 Enable HAVING clause but no fixes elsewhere yet.  
 Make "char" type a synonym for "char(1)" (actually implemented as bpchar)(Thomas)  
 Save string type if specified for DEFAULT clause handling(Thomas)  
 Coerce operations involving different data types(Thomas)  
 Allow some index use for columns of different types(Thomas)  
 Add capabilities for automatic type conversion(Thomas)  
 Cleanups for large objects, so file is truncated on open(Peter)  
 Readline cleanups(Tom)  
 Allow psql \f \ to make spaces as delimiter(Bruce)  
 Pass pg\_attribute.atttypmod to the frontend for column field lengths(Tom, Bruce)  
 Msql compatibility library in /contrib(Aldrin)  
 Remove the requirement that ORDER/GROUP BY clause identifiers be included in the target list(David)  
 Convert columns to match columns in UNION clauses(Thomas)  
 Remove fork()/exec() and only do fork()(Bruce)  
 Jdbc cleanups(Peter)  
 Show backend status on ps command line(only works on some platforms)(Bruce)  
 Pg\_hba.conf now has a sameuser option in the database field  
 Make lo\_unlink take oid param, not int4  
 New DISABLE\_COMPLEX\_MACRO for compilers that can't handle our macros(Bruce)  
 Libpgtcl now handles NOTIFY as a Tcl event, need not send dummy queries(Tom)  
 libpgtcl cleanups(Tom)  
 Add -error option to libpgtcl's pg\_result command(Tom)  
 New locale patch, see docs/README/locale(Oleg)

Fix for pg\_dump so CONSTRAINT and CHECK syntax is correct(ccb)  
 New contrib/lo code for large object orphan removal(Peter)  
 New psql command "SET CLIENT\_ENCODING TO 'encoding'" for multibytes  
 feature, see /doc/README.mb(Tatsuo)  
 contrib/noupdate code to revoke update permission on a column  
 libpq can now be compiled on Windows(Magnus)  
 Add PQsetdbLogin() in libpq  
 New 8-byte integer type, checked by configure for OS support(Thomas)  
 Better support for quoted table/column names(Thomas)  
 Surround table and column names with double-quotes in pg\_dump(Thomas)  
 PQreset() now works with passwords(Tom)  
 Handle case of GROUP BY target list column number out of range(David)  
 Allow UNION in subselects  
 Add auto-size to screen to \d? commands(Bruce)  
 Use UNION to show all \d? results in one query(Bruce)  
 Add \d? field search feature(Bruce)  
 Pg\_dump issues fewer \connect requests(Tom)  
 Make pg\_dump -z flag work better, document it in manual page(Tom)  
 Add HAVING clause with full support for subselects and unions(Stephan)  
 Full text indexing routines in contrib/fulltextindex(Maarten)  
 Transaction ids now stored in shared memory(Vadim)  
 New PGCLIENTENCODING when issuing COPY command(Tatsuo)  
 Support for SQL92 syntax "SET NAMES"(Tatsuo)  
 Support for LATIN2-5(Tatsuo)  
 Add UNICODE regression test case(Tatsuo)  
 Lock manager cleanup, new locking modes for LLL(Vadim)  
 Allow index use with OR clauses(Bruce)  
 Allows "SELECT NULL ORDER BY 1;"  
 Explain VERBOSE prints the plan, and now pretty-prints the plan to  
 the postmaster log file(Bruce)  
 Add indexes display to \d command(Bruce)  
 Allow GROUP BY on functions(David)  
 New pg\_class.relkind for large objects(Bruce)  
 New way to send libpq NOTICE messages to a different location(Tom)  
 New \w write command to psql(Bruce)  
 New /contrib/findoidjoins scans oid columns to find join relationships(Bruce)  
 Allow binary-compatible indexes to be considered when checking for valid  
 Indexes for restriction clauses containing a constant(Thomas)  
 New ISBN/ISSN code in /contrib/isbn\_issn  
 Allow NOT LIKE, IN, NOT IN, BETWEEN, and NOT BETWEEN constraint(Thomas)  
 New rewrite system fixes many problems with rules and views(Jan)
 

- \* Rules on relations work
- \* Event qualifications on insert/update/delete work
- \* New OLD variable to reference CURRENT, CURRENT will be remove in future
- \* Update rules can reference NEW and OLD in rule qualifications/actions
- \* Insert/update/delete rules on views work
- \* Multiple rule actions are now supported, surrounded by parentheses
- \* Regular users can create views/rules on tables they have RULE permits
- \* Rules and views inherit the privileges of the creator
- \* No rules at the column level
- \* No UPDATE NEW/OLD rules
- \* New pg\_tables, pg\_indexes, pg\_rules and pg\_views system views
- \* Only a single action on SELECT rules
- \* Total rewrite overhaul, perhaps for 6.5
- \* handle subselects
- \* handle aggregates on views
- \* handle insert into select from view works

 System indexes are now multikey(Bruce)  
 Oidint2, oidint4, and oidname types are removed(Bruce)  
 Use system cache for more system table lookups(Bruce)  
 New backend programming language PL/pgSQL in backend/pl(Jan)  
 New SERIAL data type, auto-creates sequence/index(Thomas)

Enable assert checking without a recompile(Massimo)  
 User lock enhancements(Massimo)  
 New setval() command to set sequence value(Massimo)  
 Auto-remove unix socket file on start-up if no postmaster running(Massimo)  
 Conditional trace package(Massimo)  
 New UNLISTEN command(Massimo)  
 psql and libpq now compile under Windows using win32.mak(Magnus)  
 Lo\_read no longer stores trailing NULL(Bruce)  
 Identifiers are now truncated to 31 characters internally(Bruce)  
 Createuser options now available on the command line  
 Code for 64-bit integer supported added, configure tested, int8 type(Thomas)  
 Prevent file descriptor leak from failed COPY(Bruce)  
 New pg\_upgrade command(Bruce)  
 Updated /contrib directories(Massimo)  
 New CREATE TABLE DEFAULT VALUES statement available(Thomas)  
 New INSERT INTO TABLE DEFAULT VALUES statement available(Thomas)  
 New DECLARE and FETCH feature(Thomas)  
 libpq's internal structures now not exported(Tom)  
 Allow up to 8 key indexes(Bruce)  
 Remove ARCHIVE key word, that is no longer used(Thomas)  
 pg\_dump -n flag to suppress quotes around identifiers  
 disable system columns for views(Jan)  
 new INET and CIDR types for network addresses(TomH, Paul)  
 no more double quotes in psql output  
 pg\_dump now dumps views(Terry)  
 new SET QUERY\_LIMIT(Tatsuo, Jan)

#### Source Tree Changes

-----  
 /contrib cleanup(Jun)  
 Inline some small functions called for every row(Bruce)  
 Alpha/linux fixes  
 HP-UX cleanups(Tom)  
 Multibyte regression tests(Soonmyung.)  
 Remove --disabled options from configure  
 Define PGDOC to use POSTGRES DIR by default  
 Make regression optional  
 Remove extra braces code to pgindent(Bruce)  
 Add bsdi shared library support(Bruce)  
 New --without-CXX support configure option(Brook)  
 New FAQ\_CVS  
 Update backend flowchart in tools/backend(Bruce)  
 Change atttypmod from int16 to int32(Bruce, Tom)  
 Getrusage() fix for platforms that do not have it(Tom)  
 Add PQconnectdb, PGUSER, PGPASSWORD to libpq man page  
 NS32K platform fixes(Phil Nelson, John Buller)  
 SCO 7/UnixWare 2.x fixes(Billy, others)  
 Sparc/Solaris 2.5 fixes(Ryan)  
 Pgbuiltin.3 is obsolete, move to doc files(Thomas)  
 Even more documentation(Thomas)  
 Nextstep support(Jacek)  
 Aix support(David)  
 pginterface manual page(Bruce)  
 shared libraries all have version numbers  
 merged all OS-specific shared library defines into one file  
 smarter TCL/Tk configuration checking(Billy)  
 smarter perl configuration(Brook)  
 configure uses supplied install-sh if no install script found(Tom)  
 new Makefile.shlib for shared library configuration(Tom)

## E.40. Release 6.3.2

**Release date:** 1998-04-07

This is a bug-fix release for 6.3.x. Refer to the release notes for version 6.3 for a more complete summary of new features.

Summary:

- Repairs automatic configuration support for some platforms, including Linux, from breakage inadvertently introduced in version 6.3.1.
- Correctly handles function calls on the left side of BETWEEN and LIKE clauses.

A dump/restore is NOT required for those running 6.3 or 6.3.1. A `make distclean`, `make`, and `make install` is all that is required. This last step should be performed while the postmaster is not running. You should re-link any custom applications that use PostgreSQL libraries.

For upgrades from pre-6.3 installations, refer to the installation and migration instructions for version 6.3.

### E.40.1. Modificações

Configure detection improvements for tcl/tk(Brook Milligan, Alvin)  
 Manual page improvements(Bruce)  
 BETWEEN and LIKE fix(Thomas)  
 fix for psql \connect used by pg\_dump(Oliver Elphick)  
 New odbc driver  
 pgaccess, version 0.86  
 qsort removed, now uses libc version, cleanups(Jeroen)  
 fix for buffer over-runs detected(Maurice Gittens)  
 fix for buffer overrun in libpgtcl(Randy Kunkee)  
 fix for UNION with DISTINCT or ORDER BY(Bruce)  
 gettimeofday configure check(Doug Winterburn)  
 Fix "indexes not used" bug(Vadim)  
 docs additions(Thomas)  
 Fix for backend memory leak(Bruce)  
 libreadline cleanup(Erwan MAS)  
 Remove DISTDIR(Bruce)  
 Makefile dependency cleanup(Jeroen van Vianen)  
 ASSERT fixes(Bruce)

## E.41. Release 6.3.1

**Release date:** 1998-03-23

Summary:

- Additional support for multibyte character sets.
- Repair byte ordering for mixed-endian clients and servers.
- Minor updates to allowed SQL syntax.
- Improvements to the configuration autodetection for installation.

A dump/restore is NOT required for those running 6.3. A `make distclean`, `make`, and `make install` is all that is required. This last step should be performed while the postmaster is not running. You should re-link any custom applications that use PostgreSQL libraries.

For upgrades from pre-6.3 installations, refer to the installation and migration instructions for version 6.3.

### E.41.1. Modificações

ecpg cleanup/fixes, now version 1.1(Michael Meskes)  
 pg\_user cleanup(Bruce)  
 large object fix for pg\_dump and tclsh (alvin)

LIKE fix for multiple adjacent underscores  
 fix for redefining builtin functions(Thomas)  
 ultrix4 cleanup  
 upgrade to pg\_access 0.83  
 updated CLUSTER manual page  
 multibyte character set support, see doc/README.mb(Tatsuo)  
 configure --with-pgport fix  
 pg\_ident fix  
 big-endian fix for backend communications(Kataoka)  
 SUBSTR() and substring() fix(Jan)  
 several jdbc fixes(Peter)  
 libpgtcl improvements, see libpgtcl/README(Randy Kunkee)  
 Fix for "Datasize = 0" error(Vadim)  
 Prevent \do from wrapping(Bruce)  
 Remove duplicate Russian character set entries  
 Sunos4 cleanup  
 Allow optional TABLE key word in LOCK and SELECT INTO(Thomas)  
 CREATE SEQUENCE options to allow a negative integer(Thomas)  
 Add "PASSWORD" as an allowed column identifier(Thomas)  
 Add checks for UNION target fields(Bruce)  
 Fix Alpha port(Dwayne Bailey)  
 Fix for text arrays containing quotes(Doug Gibson)  
 Solaris compile fix(Albert Chin-A-Young)  
 Better identify tcl and tk libs and includes(Bruce)

## E.42. Release 6.3

**Release date:** 1998-03-01

There are *many* new features and improvements in this release. Here is a brief, incomplete summary:

- Many new SQL features, including full SQL92 subselect capability (everything is here but target-list subselects).
- Support for client-side environment variables to specify time zone and date style.
- Socket interface for client/server connection. This is the default now so you may need to start postmaster with the `-i` flag.
- Better password authorization mechanisms. Default table privileges have changed.
- Old-style *time travel* has been removed. Performance has been improved.

**Nota:** Bruce Momjian wrote the following notes to introduce the new release.

There are some general 6.3 issues that I want to mention. These are only the big items that can not be described in one sentence. A review of the detailed changes list is still needed.

First, we now have subselects. Now that we have them, I would like to mention that without subselects, SQL is a very limited language. Subselects are a major feature, and you should review your code for places where subselects provide a better solution for your queries. I think you will find that there are more uses for subselects than you may think. Vadim has put us on the big SQL map with subselects, and fully functional ones too. The only thing you can't do with subselects is to use them in the target list.

Second, 6.3 uses Unix domain sockets rather than TCP/IP by default. To enable connections from other machines, you have to use the new postmaster `-i` option, and of course edit `pg_hba.conf`. Also, for this reason, the format of `pg_hba.conf` has changed.

Third, `char()` fields will now allow faster access than `varchar()` or `text`. Specifically, the `text` and `varchar()` have a penalty for access to any columns after the first column of this type. `char()` used to also have this access penalty, but it no longer does. This may suggest that you redesign some of your tables, especially if you have short character columns that you have defined as `varchar()` or `text`. This and other changes make 6.3 even faster than earlier releases.

We now have passwords definable independent of any Unix file. There are new SQL USER commands. See the *Administrator's Guide* for more information. There is a new table, `pg_shadow`, which is used to store user information and user passwords, and it by default only SELECT-able by the postgres super-user. `pg_user` is now a view of `pg_shadow`, and is SELECT-able by PUBLIC. You should keep using `pg_user` in your application without changes.

User-created tables now no longer have SELECT privilege to PUBLIC by default. This was done because the ANSI standard requires it. You can of course GRANT any privileges you want after the table is created. System tables continue to be SELECT-able by PUBLIC.

We also have real deadlock detection code. No more sixty-second timeouts. And the new locking code implements a FIFO better, so there should be less resource starvation during heavy use.

Many complaints have been made about inadequate documentation in previous releases. Thomas has put much effort into many new manuals for this release. Check out the `doc/` directory.

For performance reasons, time travel is gone, but can be implemented using triggers (see `pgsql/contrib/spi/README`). Please check out the new `\d` command for types, operators, etc. Also, views have their own privileges now, not based on the underlying tables, so privileges on them have to be set separately. Check `/pgsql/interfaces` for some new ways to talk to PostgreSQL.

This is the first release that really required an explanation for existing users. In many ways, this was necessary because the new release removes many limitations, and the work-arounds people were using are no longer needed.

### E.42.1. Migration to version 6.3

A dump/restore using `pg_dump` or `pg_dumpall` is required for those wishing to migrate data from any previous release of PostgreSQL.

### E.42.2. Modificações

#### Bug Fixes

-----

```
Fix binary cursors broken by MOVE implementation(Vadim)
Fix for tcl library crash(Jan)
Fix for array handling, from Gerhard Hintermayer
Fix acl error, and remove duplicate pqtrace(Bruce)
Fix psql \e for empty file(Bruce)
Fix for textcat on varchar() fields(Bruce)
Fix for DBT Sendproc (Zeugswetter Andres)
Fix vacuum analyze syntax problem(Bruce)
Fix for international identifiers(Tatsuo)
Fix aggregates on inherited tables(Bruce)
Fix substr() for out-of-bounds data
Fix for select 1=1 or 2=2, select 1=1 and 2=2, and select sum(2+2)(Bruce)
Fix notty output to show status result. -q option still turns it off(Bruce)
Fix for count(*), aggs with views and multiple tables and sum(3)(Bruce)
Fix cluster(Bruce)
Fix for PQtrace start/stop several times(Bruce)
Fix a variety of locking problems like newer lock waiters getting
    lock before older waiters, and having readlock people not share
    locks if a writer is waiting for a lock, and waiting writers not
    getting priority over waiting readers(Bruce)
Fix crashes in psql when executing queries from external files(James)
Fix problem with multiple order by columns, with the first one having
    NULL values(Jeroen)
Use correct hash table support functions for float8 and int4(Thomas)
Re-enable JOIN= option in CREATE OPERATOR statement (Thomas)
Change precedence for boolean operators to match expected behavior(Thomas)
Generate elog(ERROR) on over-large integer(Bruce)
Allow multiple-argument functions in constraint clauses(Thomas)
Check boolean input literals for 'true', 'false', 'yes', 'no', '1', '0'
    and throw elog(ERROR) if unrecognized(Thomas)
Major large objects fix
```

Fix for GROUP BY showing duplicates(Vadim)  
 Fix for index scans in MergeJoin(Vadim)

## Enhancements

-----

Subselects with EXISTS, IN, ALL, ANY key words (Vadim, Bruce, Thomas)  
 New User Manual(Thomas, others)  
 Speedup by inlining some frequently-called functions  
 Real deadlock detection, no more timeouts(Bruce)  
 Add SQL92 "constants" CURRENT\_DATE, CURRENT\_TIME, CURRENT\_TIMESTAMP,  
     CURRENT\_USER(Thomas)  
 Modify constraint syntax to be SQL92-compliant(Thomas)  
 Implement SQL92 PRIMARY KEY and UNIQUE clauses using indexes(Thomas)  
 Recognize SQL92 syntax for FOREIGN KEY. Throw elog notice(Thomas)  
 Allow NOT NULL UNIQUE constraint clause (each allowed separately before)(Thomas)  
 Allow PostgreSQL-style casting ("::") of non-constants(Thomas)  
 Add support for SQL3 TRUE and FALSE boolean constants(Thomas)  
 Support SQL92 syntax for IS TRUE/IS FALSE/IS NOT TRUE/IS NOT FALSE(Thomas)  
 Allow shorter strings for boolean literals (e.g. "t", "tr", "tru")(Thomas)  
 Allow SQL92 delimited identifiers(Thomas)  
 Implement SQL92 binary and hexadecimal string decoding (b'10' and x'1F')(Thomas)  
 Support SQL92 syntax for type coercion of literal strings  
     (e.g. "DATETIME 'now'")(Thomas)  
 Add conversions for int2, int4, and OID types to and from text(Thomas)  
 Use shared lock when building indexes(Vadim)  
 Free memory allocated for an user query inside transaction block after  
     this query is done, was turned off in <= 6.2.1(Vadim)  
 New SQL statement CREATE PROCEDURAL LANGUAGE(Jan)  
 New PostgreSQL Procedural Language (PL) backend interface(Jan)  
 Rename pg\_dump -H option to -h(Bruce)  
 Add Java support for passwords, European dates(Peter)  
 Use indexes for LIKE and ~, !~ operations(Bruce)  
 Add hash functions for datetime and timespan(Thomas)  
 Time Travel removed(Vadim, Bruce)  
 Add paging for \d and \z, and fix \i(Bruce)  
 Add Unix domain socket support to backend and to frontend library(Goran)  
 Implement CREATE DATABASE/WITH LOCATION and initlocation utility(Thomas)  
 Allow more SQL92 and/or PostgreSQL reserved words as column identifiers(Thomas)  
 Augment support for SQL92 SET TIME ZONE...(Thomas)  
 SET/SHOW/RESET TIME ZONE uses TZ backend environment variable(Thomas)  
 Implement SET keyword = DEFAULT and SET TIME ZONE DEFAULT(Thomas)  
 Enable SET TIME ZONE using TZ environment variable(Thomas)  
 Add PGDATESTYLE environment variable to frontend and backend initialization(Thomas)  
 Add PGTZ, PGCSOHEAP, PGCSOINDEX, PGRPLANS, PGGEQO  
     frontend library initialization environment variables(Thomas)  
 Regression tests time zone automatically set with "setenv PGTZ PST8PDT"(Thomas)  
 Add pg\_description table for info on tables, columns, operators, types, and  
     aggregates(Bruce)  
 Increase 16 char limit on system table/index names to 32 characters(Bruce)  
 Rename system indexes(Bruce)  
 Add 'GERMAN' option to SET DATESTYLE(Thomas)  
 Define an "ISO-style" timespan output format with "hh:mm:ss" fields(Thomas)  
 Allow fractional values for delta times (e.g. '2.5 days')(Thomas)  
 Validate numeric input more carefully for delta times(Thomas)  
 Implement day of year as possible input to date\_part()(Thomas)  
 Define timespan\_finite() and text\_timespan() functions(Thomas)  
 Remove archive stuff(Bruce)  
 Allow for a pg\_password authentication database that is separate from  
     the system password file(Todd)  
 Dump ACLs, GRANT, REVOKE privileges(Matt)  
 Define text, varchar, and bpchar string length functions(Thomas)  
 Fix Query handling for inheritance, and cost computations(Bruce)



Implement CREATE TABLE/AS SELECT (alternative to SELECT/INTO)(Thomas)  
 Allow NOT, IS NULL, IS NOT NULL in constraints(Thomas)  
 Implement UNIONs for SELECT(Bruce)  
 Add UNION, GROUP, DISTINCT to INSERT(Bruce)  
 varchar() stores only necessary bytes on disk(Bruce)  
 Fix for BLOBs(Peter)  
 Mega-Patch for JDBC...see README\_6.3 for list of changes(Peter)  
 Remove unused "option" from PQconnectdb()  
 New LOCK command and lock manual page describing deadlocks(Bruce)  
 Add new psql \da, \dd, \df, \do, \dS, and \dT commands(Bruce)  
 Enhance psql \z to show sequences(Bruce)  
 Show NOT NULL and DEFAULT in psql \d table(Bruce)  
 New psql .psqlrc file start-up(Andrew)  
 Modify sample start-up script in contrib/linux to show syslog(Thomas)  
 New types for IP and MAC addresses in contrib/ip\_and\_mac(TomH)  
 Unix system time conversions with date/time types in contrib/unixdate(Thomas)  
 Update of contrib stuff(Massimo)  
 Add Unix socket support to DBD::Pg(Goran)  
 New python interface (PyGreSQL 2.0)(D'Arcy)  
 New frontend/backend protocol has a version number, network byte order(Phil)  
 Security features in pg\_hba.conf enhanced and documented, many cleanups(Phil)  
 CHAR() now faster access than VARCHAR() or TEXT  
 ecpg embedded SQL preprocessor  
 Reduce system column overhead(Vadmin)  
 Remove pg\_time table(Vadim)  
 Add pg\_type attribute to identify types that need length (bpchar, varchar)  
 Add report of offending line when COPY command fails  
 Allow VIEW privileges to be set separately from the underlying tables.  
     For security, use GRANT/REVOKE on views as appropriate(Jan)  
 Tables now have no default GRANT SELECT TO PUBLIC. You must  
     explicitly grant such privileges.  
 Clean up tutorial examples(Darren)

#### Source Tree Changes

-----  
 Add new html development tools, and flow chart in /tools/backend  
 Fix for SCO compiles  
 Stratus computer port Robert Gillies  
 Added support for shlib for BSD44\_derived & i386\_solaris  
 Make configure more automated(Brook)  
 Add script to check regression test results  
 Break parser functions into smaller files, group together(Bruce)  
 Rename heap\_create to heap\_create\_and\_catalog, rename heap\_creatr  
     to heap\_create()(Bruce)  
 Sparc/Linux patch for locking(TomS)  
 Remove PORTNAME and reorganize port-specific stuff(Marc)  
 Add optimizer README file(Bruce)  
 Remove some recursion in optimizer and clean up some code there(Bruce)  
 Fix for NetBSD locking(Henry)  
 Fix for libptcl make(Tatsuo)  
 AIX patch(Darren)  
 Change IS TRUE, IS FALSE, ... to expressions using "=" rather than  
     function calls to istru() or isfalse() to allow optimization(Thomas)  
 Various fixes NetBSD/Sparc related(TomH)  
 Alpha linux locking(Travis,Ryan)  
 Change elog(WARN) to elog(ERROR)(Bruce)  
 FAQ for FreeBSD(Marc)  
 Bring in the PostODBC source tree as part of our standard distribution(Marc)  
 A minor patch for HP/UX 10 vs 9(Stan)  
 New pg\_attribute.atttypmod for type-specific info like varchar length(Bruce)  
 UnixWare patches(Billy)  
 New i386 'lock' for spinlock asm(Billy)

Support for multiplexed backends is removed  
 Start an OpenBSD port  
 Start an AUX port  
 Start a Cygnus port  
 Add string functions to regression suite(Thomas)  
 Expand a few function names formerly truncated to 16 characters(Thomas)  
 Remove un-needed malloc() calls and replace with palloc()(Bruce)

## E.43. Release 6.2.1

**Release date:** 1997-10-17

6.2.1 is a bug-fix and usability release on 6.2.

Summary:

- Allow strings to span lines, per SQL92.
- Include example trigger function for inserting user names on table updates.

This is a minor bug-fix release on 6.2. For upgrades from pre-6.2 systems, a full dump/reload is required. Refer to the 6.2 release notes for instructions.

### E.43.1. Migration from version 6.2 to version 6.2.1

This is a minor bug-fix release. A dump/reload is not required from version 6.2, but is required from any release prior to 6.2.

In upgrading from version 6.2, if you choose to dump/reload you will find that avg(money) is now calculated correctly. All other bug fixes take effect upon updating the executables.

Another way to avoid dump/reload is to use the following SQL command from `psql` to update the existing system table:

```
update pg_aggregate set aggfinalfn = 'cash_div_flt8'
where aggrname = 'avg' and aggbasetype = 790;
```

This will need to be done to every existing database, including `template1`.

### E.43.2. Modificações

Allow TIME and TYPE column names(Thomas)  
 Allow larger range of true/false as boolean values(Thomas)  
 Support output of "now" and "current"(Thomas)  
 Handle DEFAULT with INSERT of NULL properly(Vadim)  
 Fix for relation reference counts problem in buffer manager(Vadim)  
 Allow strings to span lines, like ANSI(Thomas)  
 Fix for backward cursor with ORDER BY(Vadim)  
 Fix avg(cash) computation(Thomas)  
 Fix for specifying a column twice in ORDER/GROUP BY(Vadim)  
 Documented new libpq function to return affected rows, PQcmdTuples(Bruce)  
 Trigger function for inserting user names for INSERT/UPDATE(Brook Milligan)

## E.44. Release 6.2

**Release date:** 1997-10-02

A dump/restore is required for those wishing to migrate data from previous releases of PostgreSQL.

### E.44.1. Migration from version 6.1 to version 6.2

This migration requires a complete dump of the 6.1 database and a restore of the database in 6.2.

Note that the `pg_dump` and `pg_dumpall` utility from 6.2 should be used to dump the 6.1 database.

## E.44.2. Migration from version 1.x to version 6.2

Those migrating from earlier 1.\* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

## E.44.3. Modificações

### Bug Fixes

```

-----
Fix problems with pg_dump for inheritance, sequences, archive tables(Bruce)
Fix compile errors on overflow due to shifts, unsigned, and bad prototypes
    from Solaris(Diab Jerius)
Fix bugs in geometric line arithmetic (bad intersection calculations)(Thomas)
Check for geometric intersections at endpoints to avoid rounding ugliness(Thomas)
Catch non-functional delete attempts(Vadim)
Change time function names to be more consistent(Michael Reifenberg)
Check for zero divides(Michael Reifenberg)
Fix very old bug which made rows changed/inserted by a command
    visible to the command itself (so we had multiple update of
    updated rows, etc.)(Vadim)
Fix for SELECT null, 'fail' FROM pg_am (Patrick)
SELECT NULL as EMPTY_FIELD now allowed(Patrick)
Remove un-needed signal stuff from contrib/pginterface
Fix OR (where x != 1 or x isnull didn't return rows with x NULL) (Vadim)
Fix time_cmp function (Vadim)
Fix handling of functions with non-attribute first argument in
    WHERE clauses (Vadim)
Fix GROUP BY when order of entries is different from order
    in target list (Vadim)
Fix pg_dump for aggregates without sfunc1 (Vadim)

```

### Enhancements

```

-----
Default genetic optimizer GEQO parameter is now 8(Bruce)
Allow use parameters in target list having aggregates in functions(Vadim)
Added JDBC driver as an interface(Adrian & Peter)
pg_password utility
Return number of rows inserted/affected by INSERT/UPDATE/DELETE etc.(Vadim)
Triggers implemented with CREATE TRIGGER (SQL3)(Vadim)
SPI (Server Programming Interface) allows execution of queries inside
    C-functions (Vadim)
NOT NULL implemented (SQL92)(Robson Paniago de Miranda)
Include reserved words for string handling, outer joins, and unions(Thomas)
Implement extended comments ("/* ... */") using exclusive states(Thomas)
Add "//" single-line comments(Bruce)
Remove some restrictions on characters in operator names(Thomas)
DEFAULT and CONSTRAINT for tables implemented (SQL92)(Vadim & Thomas)
Add text concatenation operator and function (SQL92)(Thomas)
Support WITH TIME ZONE syntax (SQL92)(Thomas)
Support INTERVAL unit TO unit syntax (SQL92)(Thomas)
Define types DOUBLE PRECISION, INTERVAL, CHARACTER,
    and CHARACTER VARYING (SQL92)(Thomas)
Define type FLOAT(p) and rudimentary DECIMAL(p,s), NUMERIC(p,s) (SQL92)(Thomas)
Define EXTRACT(), POSITION(), SUBSTRING(), and TRIM() (SQL92)(Thomas)
Define CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP (SQL92)(Thomas)
Add syntax and warnings for UNION, HAVING, INNER and OUTER JOIN (SQL92)(Thomas)
Add more reserved words, mostly for SQL92 compliance(Thomas)
Allow hh:mm:ss time entry for timespan/reftime types(Thomas)
Add center() routines for lseg, path, polygon(Thomas)
Add distance() routines for circle-polygon, polygon-polygon(Thomas)
Check explicitly for points and polygons contained within polygons
    using an axis-crossing algorithm(Thomas)

```

Add routine to convert circle-box(Thomas)  
Merge conflicting operators for different geometric data types(Thomas)  
Replace distance operator "<==>" with "<->"(Thomas)  
Replace "above" operator "!^" with ">^" and "below" operator "!" with "<^"(Thomas)  
Add routines for text trimming on both ends, substring, and string position(Thomas)  
Added conversion routines circle(box) and poly(circle)(Thomas)  
Allow internal sorts to be stored in memory rather than in files(Bruce & Vadim)  
Allow functions and operators on internally-identical types to succeed(Bruce)  
Speed up backend start-up after profiling analysis(Bruce)  
Inline frequently called functions for performance(Bruce)  
Reduce open() calls(Bruce)  
psql: Add PAGER for \h and \?, \C fix  
Fix for psql pager when no tty(Bruce)  
New entab utility(Bruce)  
General trigger functions for referential integrity (Vadim)  
General trigger functions for time travel (Vadim)  
General trigger functions for AUTOINCREMENT/IDENTITY feature (Vadim)  
MOVE implementation (Vadim)

#### Source Tree Changes

-----  
HP-UX 10 patches (Vladimir Turin)  
Added SCO support, (Daniel Harris)  
MkLinux patches (Tatsuo Ishii)  
Change geometric box terminology from "length" to "width"(Thomas)  
Deprecate temporary unstored slope fields in geometric code(Thomas)  
Remove restart instructions from INSTALL(Bruce)  
Look in /usr/ucb first for install(Bruce)  
Fix c++ copy example code(Thomas)  
Add -o to psql manual page(Bruce)  
Prevent relname unallocated string length from being copied into database(Bruce)  
Cleanup for NAMEDATALEN use(Bruce)  
Fix pg\_proc names over 15 chars in output(Bruce)  
Add strNcpy() function(Bruce)  
remove some (void) casts that are unnecessary(Bruce)  
new interfaces directory(Marc)  
Replace fopen() calls with calls to fd.c functions(Bruce)  
Make functions static where possible(Bruce)  
enclose unused functions in #ifdef NOT\_USED(Bruce)  
Remove call to difftime() in timestamp support to fix SunOS(Bruce & Thomas)  
Changes for Digital Unix  
Portability fix for pg\_dumpall(Bruce)  
Rename pg\_attribute.attnvals to attdispersion(Bruce)  
"intro/unix" manual page now "pgintro"(Bruce)  
"built-in" manual page now "pgbuiltin"(Bruce)  
"drop" manual page now "drop\_table"(Bruce)  
Add "create\_trigger", "drop\_trigger" manual pages(Thomas)  
Add constraints regression test(Vadim & Thomas)  
Add comments syntax regression test(Thomas)  
Add PGINDENT and support program(Bruce)  
Massive commit to run PGINDENT on all \*.c and \*.h files(Bruce)  
Files moved to /src/tools directory(Bruce)  
SPI and Trigger programming guides (Vadim & D'Arcy)

## E.45. Release 6.1.1

Release date: 1997-07-22

### E.45.1. Migration from version 6.1 to version 6.1.1

This is a minor bug-fix release. A dump/reload is not required from version 6.1, but is required from any release prior to 6.1. Refer to the release notes for 6.1 for more details.

### E.45.2. Modificações

```
fix for SET with options (Thomas)
allow pg_dump/pg_dumpall to preserve ownership of all tables/objects(Bruce)
new psql \connect option allows changing usernames without changing databases
fix for initdb --debug option(Yoshihiko Ichikawa)
lextest cleanup(Bruce)
hash fixes(Vadim)
fix date/time month boundary arithmetic(Thomas)
fix timezone daylight handling for some ports(Thomas, Bruce, Tatsuo)
timestamp overhauled to use standard functions(Thomas)
other code cleanup in date/time routines(Thomas)
psql's \d now case-insensitive(Bruce)
psql's backslash commands can now have trailing semicolon(Bruce)
fix memory leak in psql when using \g(Bruce)
major fix for endian handling of communication to server(Thomas, Tatsuo)
Fix for Solaris assembler and include files(Yoshihiko Ichikawa)
allow underscores in usernames(Bruce)
pg_dumpall now returns proper status, portability fix(Bruce)
```

## E.46. Release 6.1

**Release date:** 1997-06-08

The regression tests have been adapted and extensively modified for the 6.1 release of PostgreSQL.

Three new data types (`datetime`, `timespan`, and `circle`) have been added to the native set of PostgreSQL types. Points, boxes, paths, and polygons have had their output formats made consistent across the data types. The polygon output in `misc.out` has only been spot-checked for correctness relative to the original regression output.

PostgreSQL 6.1 introduces a new, alternate optimizer which uses *genetic* algorithms. These algorithms introduce a random behavior in the ordering of query results when the query contains multiple qualifiers or multiple tables (giving the optimizer a choice on order of evaluation). Several regression tests have been modified to explicitly order the results, and hence are insensitive to optimizer choices. A few regression tests are for data types which are inherently unordered (e.g. points and time intervals) and tests involving those types are explicitly bracketed with `set geqo to 'off'` and `reset geqo`.

The interpretation of array specifiers (the curly braces around atomic values) appears to have changed sometime after the original regression tests were generated. The current `./expected/*.out` files reflect this new interpretation, which may not be correct!

The float8 regression test fails on at least some platforms. This is due to differences in implementations of `pow()` and `exp()` and the signaling mechanisms used for overflow and underflow conditions.

The “random” results in the random test should cause the “random” test to be “failed”, since the regression tests are evaluated using a simple diff. However, “random” does not seem to produce random results on my test machine (Linux/gcc/i686).

### E.46.1. Migration to version 6.1

This migration requires a complete dump of the 6.0 database and a restore of the database in 6.1.

Those migrating from earlier 1.\* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

### E.46.2. Modificações

## Bug Fixes

-----

packet length checking in library routines  
 lock manager priority patch  
 check for under/over flow of float8(Bruce)  
 multitable join fix(Vadim)  
 SIGPIPE crash fix(Darren)  
 large object fixes(Sven)  
 allow btree indexes to handle NULLs(Vadim)  
 timezone fixes(D'Arcy)  
 select SUM(x) can return NULL on no rows(Thomas)  
 internal optimizer, executor bug fixes(Vadim)  
 fix problem where inner loop in < or <= has no rows(Vadim)  
 prevent re-commuting join index clauses(Vadim)  
 fix join clauses for multiple tables(Vadim)  
 fix hash, hashjoin for arrays(Vadim)  
 fix btree for abstime type(Vadim)  
 large object fixes(Raymond)  
 fix buffer leak in hash indexes (Vadim)  
 fix rtree for use in inner scan (Vadim)  
 fix gist for use in inner scan, cleanups (Vadim, Andrea)  
 avoid unnecessary local buffers allocation (Vadim, Massimo)  
 fix local buffers leak in transaction aborts (Vadim)  
 fix file manager memory leaks, cleanups (Vadim, Massimo)  
 fix storage manager memory leaks (Vadim)  
 fix btree duplicates handling (Vadim)  
 fix deleted rows reincarnation caused by vacuum (Vadim)  
 fix SELECT varchar()/char() INTO TABLE made zero-length fields(Bruce)  
 many psql, pg\_dump, and libpq memory leaks fixed using Purify (Igor)

## Enhancements

-----

attribute optimization statistics(Bruce)  
 much faster new btree bulk load code(Paul)  
 BTREE UNIQUE added to bulk load code(Vadim)  
 new lock debug code(Massimo)  
 massive changes to libpg++(Leo)  
 new GEQO optimizer speeds table multitable optimization(Martin)  
 new WARN message for non-unique insert into unique key(Marc)  
 update x=-3, no spaces, now valid(Bruce)  
 remove case-sensitive identifier handling(Bruce,Thomas,Dan)  
 debug backend now pretty-prints tree(Darren)  
 new Oracle character functions(Edmund)  
 new plaintext password functions(Dan)  
 no such class or insufficient privilege changed to distinct messages(Dan)  
 new ANSI timestamp function(Dan)  
 new ANSI Time and Date types (Thomas)  
 move large chunks of data in backend(Martin)  
 multicolumn btree indexes(Vadim)  
 new SET var TO value command(Martin)  
 update transaction status on reads(Dan)  
 new locale settings for character types(Oleg)  
 new SEQUENCE serial number generator(Vadim)  
 GROUP BY function now possible(Vadim)  
 re-organize regression test(Thomas,Marc)  
 new optimizer operation weights(Vadim)  
 new psql \z grant/permit option(Marc)  
 new MONEY data type(D'Arcy,Thomas)  
 tcp socket communication speed improved(Vadim)  
 new VACUUM option for attribute statistics, and for certain columns (Vadim)  
 many geometric type improvements(Thomas,Keith)  
 additional regression tests(Thomas)

```

new datestyle variable(Thomas,Vadim,Martin)
more comparison operators for sorting types(Thomas)
new conversion functions(Thomas)
new more compact btree format(Vadim)
allow pg_dumpall to preserve database ownership(Bruce)
new SET GEQO=# and R_PLANS variable(Vadim)
old (!GEQO) optimizer can use right-sided plans (Vadim)
typechecking improvement in SQL parser(Bruce)
new SET, SHOW, RESET commands(Thomas,Vadim)
new \connect database USER option
new destroydb -i option (Igor)
new \dt and \di psql commands (Darren)
SELECT "\n" now escapes newline (A. Duursma)
new geometry conversion functions from old format (Thomas)

```

Source tree changes

-----

```

new configuration script(Marc)
readline configuration option added(Marc)
OS-specific configuration options removed(Marc)
new OS-specific template files(Marc)
no more need to edit Makefile.global(Marc)
re-arrange include files(Marc)
nextstep patches (Gregor HOFFLEIT)
removed Windows-specific code(Bruce)
removed postmaster -e option, now only postgres -e option (Bruce)
merge duplicate library code in front/backends(Martin)
now works with eBones, international Kerberos(Jun)
more shared library support
c++ include file cleanup(Bruce)
warn about buggy flex(Bruce)
DG/UX, Ultrix, IRIX, AIX portability fixes

```

## E.47. Release 6.0

**Release date:** 1997-01-29

A dump/restore is required for those wishing to migrate data from previous releases of PostgreSQL.

### E.47.1. Migration from version 1.09 to version 6.0

This migration requires a complete dump of the 1.09 database and a restore of the database in 6.0.

### E.47.2. Migration from pre-1.09 to version 6.0

Those migrating from earlier 1.\* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

### E.47.3. Modificações

Bug Fixes

-----

```

ALTER TABLE bug - running postgres process needs to re-read table definition
Allow vacuum to be run on one table or entire database(Bruce)
Array fixes
Fix array over-runs of memory writes(Kurt)
Fix elusive btree range/non-range bug(Dan)
Fix for hash indexes on some types like time and date
Fix for pg_log size explosion
Fix permissions on lo_export()(Bruce)
Fix uninitialized reads of memory(Kurt)
Fixed ALTER TABLE ... char(3) bug(Bruce)

```

Fixed a few small memory leaks  
Fixed EXPLAIN handling of options and changed full\_path option name  
Fixed output of group acl privileges  
Memory leaks (hunt and destroy with tools like Purify(Kurt))  
Minor improvements to rules system  
NOTIFY fixes  
New asserts for run-checking  
Overhauled parser/analyze code to properly report errors and increase speed  
Pg\_dump -d now handles NULL's properly(Bruce)  
Prevent SELECT NULL from crashing server (Bruce)  
Properly report errors when INSERT ... SELECT columns did not match  
Properly report errors when insert column names were not correct  
psql \g filename now works(Bruce)  
psql fixed problem with multiple statements on one line with multiple outputs  
Removed duplicate system OIDs  
SELECT \* INTO TABLE . GROUP/ORDER BY gives unlink error if table exists(Bruce)  
Several fixes for queries that crashed the backend  
Starting quote in insert string errors(Bruce)  
Submitting an empty query now returns empty status, not just " " query(Bruce)

Enhancements  
-----

Add EXPLAIN manual page(Bruce)  
Add UNIQUE index capability(Dan)  
Add hostname/user level access control rather than just hostname and user  
Add synonym of != for <>(Bruce)  
Allow "select oid,\* from table"  
Allow BY,ORDER BY to specify columns by number, or by non-alias table.column(Bruce)  
Allow COPY from the frontend(Bryan)  
Allow GROUP BY to use alias column name(Bruce)  
Allow actual compression, not just reuse on the same page(Vadim)  
Allow installation-configuration option to auto-add all local users(Bryan)  
Allow libpq to distinguish between text value '' and null(Bruce)  
Allow non-postgres users with createdb privs to destroydb's  
Allow restriction on who can create C functions(Bryan)  
Allow restriction on who can do backend COPY(Bryan)  
Can shrink tables, pg\_time and pg\_log(Vadim & Erich)  
Change debug level 2 to print queries only, changed debug heading layout(Bruce)  
Change default decimal constant representation from float4 to float8(Bruce)  
European date format now set when postmaster is started  
Execute lowercase function names if not found with exact case  
Fixes for aggregate/GROUP processing, allow 'select sum(func(x),sum(x+y) from z'  
Gist now included in the distribution(Marc)  
Ident authentication of local users(Bryan)  
Implement BETWEEN qualifier(Bruce)  
Implement IN qualifier(Bruce)  
libpq has PQgetisnull()(Bruce)  
libpq++ improvements  
New options to initdb(Bryan)  
Pg\_dump allow dump of OIDs(Bruce)  
Pg\_dump create indexes after tables are loaded for speed(Bruce)  
Pg\_dumpall dumps all databases, and the user table  
Pginterface additions for NULL values(Bruce)  
Prevent postmaster from being run as root  
psql \h and \? is now readable(Bruce)  
psql allow backslashed, semicolons anywhere on the line(Bruce)  
psql changed command prompt for lines in query or in quotes(Bruce)  
psql char(3) now displays as (bp)char in \d output(Bruce)  
psql return code now more accurate(Bryan?)  
psql updated help syntax(Bruce)  
Re-visit and fix vacuum(Vadim)  
Reduce size of regression diffs, remove timezone name difference(Bruce)



Remove compile-time parameters to enable binary distributions(Bryan)  
 Reverse meaning of HBA masks(Bryan)  
 Secure Authentication of local users(Bryan)  
 Speed up vacuum(Vadim)  
 Vacuum now had VERBOSE option(Bruce)

#### Source tree changes

-----  
 All functions now have prototypes that are compared against the calls  
 Allow asserts to be disabled easily from Makefile.global(Bruce)  
 Change oid constants used in code to #define names  
 Decoupled sparc and solaris defines(Kurt)  
 Gcc -Wall compiles cleanly with warnings only from unfixable constructs  
 Major include file reorganization/reduction(Marc)  
 Make now stops on compile failure(Bryan)  
 Makefile restructuring(Bryan, Marc)  
 Merge bsdi\_2\_1 to bsdi(Bruce)  
 Monitor program removed  
 Name change from Postgres95 to PostgreSQL  
 New config.h file(Marc, Bryan)  
 PG\_VERSION now set to 6.0 and used by postmaster  
 Portability additions, including Ultrix, DG/UX, AIX, and Solaris  
 Reduced the number of #define's, centralized #define's  
 Remove duplicate OIDS in system tables(Dan)  
 Remove duplicate system catalog info or report mismatches(Dan)  
 Removed many os-specific #define's  
 Restructured object file generation/location(Bryan, Marc)  
 Restructured port-specific file locations(Bryan, Marc)  
 Unused/uninialized variables corrected

## E.48. Release 1.09

**Release date:** 1996-11-04

Sorry, we didn't keep track of changes from 1.02 to 1.09. Some of the changes listed in 6.0 were actually included in the 1.02.1 to 1.09 releases.

## E.49. Release 1.02

**Release date:** 1996-08-01

### E.49.1. Migration from version 1.02 to version 1.02.1

Here is a new migration file for 1.02.1. It includes the 'copy' change and a script to convert old ASCII files.

**Nota:** The following notes are for the benefit of users who want to migrate databases from Postgres95 1.01 and 1.02 to Postgres95 1.02.1.

If you are starting afresh with Postgres95 1.02.1 and do not need to migrate old databases, you do not need to read any further.

In order to upgrade older Postgres95 version 1.01 or 1.02 databases to version 1.02.1, the following steps are required:

1. Start up a new 1.02.1 postmaster
2. Add the new built-in functions and operators of 1.02.1 to 1.01 or 1.02 databases. This is done by running the new 1.02.1 server against your own 1.01 or 1.02 database and applying the queries attached at the end of the file. This can be done easily through `psql`. If your 1.01 or 1.02 database is named `testdb` and you have cut the commands from the end of this file and saved them in `addfunc.sql`:

```
% psql testdb -f addfunc.sql
```

Those upgrading 1.02 databases will get a warning when executing the last two statements in the file because they are already present in 1.02. This is not a cause for concern.

## E.49.2. Dump/Reload Procedure

If you are trying to reload a `pg_dump` or text-mode, `copy tablename to stdout` generated with a previous version, you will need to run the attached `sed` script on the ASCII file before loading it into the database. The old format used `'` as end-of-data, while `\.` is now the end-of-data marker. Also, empty strings are now loaded in as `"` rather than `NULL`. See the copy manual page for full details.

```
sed 's/^\.$/\./g' <in_file >out_file
```

If you are loading an older binary copy or non-stdout copy, there is no end-of-data character, and hence no conversion necessary.

```
-- following lines added by agc to reflect the case-insensitive
-- regexp searching for varchar (in 1.02), and bpchar (in 1.02.1)
create operator ~* (leftarg = bpchar, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = bpchar, rightarg = text, procedure = texticregexne);
create operator ~* (leftarg = varchar, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = varchar, rightarg = text, procedure = texticregexne);
```

## E.49.3. Modificações

Source code maintenance and development

- \* worldwide team of volunteers
- \* the source tree now in CVS at `ftp.ki.net`

Enhancements

- \* `psql` (and underlying `libpq` library) now has many more options for formatting output, including HTML
- \* `pg_dump` now output the schema and/or the data, with many fixes to enhance completeness.
- \* `psql` used in place of `monitor` in administration shell scripts. `monitor` to be deprecated in next release.
- \* date/time functions enhanced
- \* `NULL` insert/update/comparison fixed/enhanced
- \* TCL/TK lib and shell fixed to work with both `tcl7.4/tk4.0` and `tcl7.5/tk4.1`

Bug Fixes (almost too numerous to mention)

- \* indexes
- \* storage management
- \* check for `NULL` pointer before dereferencing
- \* Makefile fixes

New Ports

- \* added SolarisX86 port
- \* added BSD/OS 2.1 port
- \* added DG/UX port

## E.50. Release 1.01

**Release date:** 1996-02-23

### E.50.1. Migration from version 1.0 to version 1.01

The following notes are for the benefit of users who want to migrate databases from Postgres95 1.0 to Postgres95 1.01.

If you are starting afresh with Postgres95 1.01 and do not need to migrate old databases, you do not need to read any further.

In order to Postgres95 version 1.01 with databases created with Postgres95 version 1.0, the following steps are required:

1. Set the definition of `NAMEDATALEN` in `src/Makefile.global` to 16 and `OIDNAMELEN` to 20.
2. Decide whether you want to use Host based authentication.
  - a. If you do, you must create a file name `pg_hba` in your top-level data directory (typically the value of your `$PGDATA`). `src/libpq/pg_hba` shows an example syntax.
  - b. If you do not want host-based authentication, you can comment out the line
 

```
HBA = 1
```

 in `src/Makefile.global`

Note that host-based authentication is turned on by default, and if you do not take steps A or B above, the out-of-the-box 1.01 will not allow you to connect to 1.0 databases.
3. Compile and install 1.01, but DO NOT do the `initdb` step.
4. Before doing anything else, terminate your 1.0 postmaster, and backup your existing `$PGDATA` directory.
5. Set your `PGDATA` environment variable to your 1.0 databases, but set up path up so that 1.01 binaries are being used.
6. Modify the file `$PGDATA/PG_VERSION` from 5.0 to 5.1
7. Start up a new 1.01 postmaster
8. Add the new built-in functions and operators of 1.01 to 1.0 databases. This is done by running the new 1.01 server against your own 1.0 database and applying the queries attached and saving in the file `1.0_to_1.01.sql`. This can be done easily through `psql`. If your 1.0 database is name `testdb`:

```
% psql testdb -f 1.0_to_1.01.sql
```

and then execute the following commands (cut and paste from here):

```
-- add builtin functions that are new to 1.01
```

```
create function int4eqoid (int4, oid) returns bool as 'foo'
language 'internal';
create function oideqint4 (oid, int4) returns bool as 'foo'
language 'internal';
create function char2icregexeq (char2, text) returns bool as 'foo'
language 'internal';
create function char2icregexne (char2, text) returns bool as 'foo'
language 'internal';
create function char4icregexeq (char4, text) returns bool as 'foo'
language 'internal';
create function char4icregexne (char4, text) returns bool as 'foo'
language 'internal';
create function char8icregexeq (char8, text) returns bool as 'foo'
language 'internal';
create function char8icregexne (char8, text) returns bool as 'foo'
language 'internal';
create function char16icregexeq (char16, text) returns bool as 'foo'
language 'internal';
create function char16icregexne (char16, text) returns bool as 'foo'
language 'internal';
create function texticregexeq (text, text) returns bool as 'foo'
language 'internal';
create function texticregexne (text, text) returns bool as 'foo'
language 'internal';
```

```
-- add builtin functions that are new to 1.01
```

```
create operator = (leftarg = int4, rightarg = oid, procedure = int4eqoid);
create operator = (leftarg = oid, rightarg = int4, procedure = oideqint4);
create operator ~* (leftarg = char2, rightarg = text, procedure = char2icregexeq);
create operator !~* (leftarg = char2, rightarg = text, procedure = char2icregexne);
create operator ~* (leftarg = char4, rightarg = text, procedure = char4icregexeq);
```

```
create operator !~* (leftarg = char4, rightarg = text, procedure = char4icregexne);
create operator ~* (leftarg = char8, rightarg = text, procedure = char8icregexeq);
create operator !~* (leftarg = char8, rightarg = text, procedure = char8icregexne);
create operator ~* (leftarg = char16, rightarg = text, procedure = char16icregexeq);
create operator !~* (leftarg = char16, rightarg = text, procedure = char16icregexne);
create operator ~* (leftarg = text, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = text, rightarg = text, procedure = texticregexne);
```

## E.50.2. Modificações

### Incompatibilities:

- \* 1.01 is backwards compatible with 1.0 database provided the user follow the steps outlined in the MIGRATION\_from\_1.0\_to\_1.01 file. If those steps are not taken, 1.01 is not compatible with 1.0 database.

### Enhancements:

- \* added PQdisplayTuples() to libpq and changed monitor and psql to use it
- \* added NeXT port (requires SysVIPC implementation)
- \* added CAST .. AS ... syntax
- \* added ASC and DESC key words
- \* added 'internal' as a possible language for CREATE FUNCTION  
internal functions are C functions which have been statically linked into the postgres backend.
- \* a new type "name" has been added for system identifiers (table names, attribute names, etc.) This replaces the old char16 type. The of name is set by the NAMEDATALEN #define in src/Makefile.global
- \* a readable reference manual that describes the query language.
- \* added host-based access control. A configuration file (\$PGDATA/pg\_hba) is used to hold the configuration data. If host-based access control is not desired, comment out HBA=1 in src/Makefile.global.
- \* changed regex handling to be uniform use of Henry Spencer's regex code regardless of platform. The regex code is included in the distribution
- \* added functions and operators for case-insensitive regular expressions. The operators are ~\* and !~\*.
- \* pg\_dump uses COPY instead of SELECT loop for better performance

### Bug fixes:

- \* fixed an optimizer bug that was causing core dumps when functions calls were used in comparisons in the WHERE clause
- \* changed all uses of getuid to geteuid so that effective uids are used
- \* psql now returns non-zero status on errors when using -c
- \* applied public patches 1-14

## E.51. Release 1.0

**Release date:** 1995-09-05

### E.51.1. Modificações

#### Copyright change:

- \* The copyright of Postgres 1.0 has been loosened to be freely modifiable and modifiable for any purpose. Please read the COPYRIGHT file. Thanks to Professor Michael Stonebraker for making this possible.

#### Incompatibilities:

- \* date formats have to be MM-DD-YYYY (or DD-MM-YYYY if you're using EUROPEAN STYLE). This follows SQL-92 specs.
- \* "delimiters" is now a key word

#### Enhancements:

- \* sql LIKE syntax has been added

- \* copy command now takes an optional USING DELIMITER specification.  
delimiters can be any single-character string.
- \* IRIX 5.3 port has been added.  
Thanks to Paul Walmsley and others.
- \* updated pg\_dump to work with new libpq
- \* \d has been added psql  
Thanks to Keith Parks
- \* regexp performance for architectures that use POSIX regex has been  
improved due to caching of precompiled patterns.  
Thanks to Alistair Crooks
- \* a new version of libpq++  
Thanks to William Wanders

#### Bug fixes:

- \* arbitrary userids can be specified in the createuser script
- \* \c to connect to other databases in psql now works.
- \* bad pg\_proc entry for float4inc() is fixed
- \* users with usecreatedb field set can now create databases without  
having to be usesuper
- \* remove access control entries when the entry no longer has any  
privileges
- \* fixed non-portable datetimes implementation
- \* added kerberos flags to the src/backend/Makefile
- \* libpq now works with kerberos
- \* typographic errors in the user manual have been corrected.
- \* btrees with multiple index never worked, now we tell you they don't  
work when you try to use them

## E.52. Postgres95 Release 0.03

Release date: 1995-07-21

### E.52.1. Modificações

#### Incompatible changes:

- \* BETA-0.3 IS INCOMPATIBLE WITH DATABASES CREATED WITH PREVIOUS VERSIONS  
(due to system catalog changes and indexing structure changes).
- \* double-quote (") is deprecated as a quoting character for string literals;  
you need to convert them to single quotes (').
- \* name of aggregates (eg. int4sum) are renamed in accordance with the  
SQL standard (eg. sum).
- \* CHANGE ACL syntax is replaced by GRANT/REVOKE syntax.
- \* float literals (eg. 3.14) are now of type float4 (instead of float8 in  
previous releases); you might have to do typecasting if you depend on it  
being of type float8. If you neglect to do the typecasting and you assign  
a float literal to a field of type float8, you may get incorrect values  
stored!
- \* LIBPQ has been totally revamped so that frontend applications  
can connect to multiple backends
- \* the usesysid field in pg\_user has been changed from int2 to int4 to  
allow wider range of Unix user ids.
- \* the netbsd/freebsd/bsd o/s ports have been consolidated into a  
single BSD44\_derived port. (thanks to Alistair Crooks)

SQL standard-compliance (the following details changes that makes postgres95  
more compliant to the SQL-92 standard):

- \* the following SQL types are now built-in: smallint, int(eger), float, real,  
char(N), varchar(N), date and time.

The following are aliases to existing postgres types:  
smallint -> int2

```
integer, int -> int4
float, real -> float4
```

char(N) and varchar(N) are implemented as truncated text types. In addition, char(N) does blank-padding.

- \* single-quote (') is used for quoting string literals; '' (in addition to \') is supported as means of inserting a single quote in a string
- \* SQL standard aggregate names (MAX, MIN, AVG, SUM, COUNT) are used (Also, aggregates can now be overloaded, i.e. you can define your own MAX aggregate to take in a user-defined type.)
- \* CHANGE ACL removed. GRANT/REVOKE syntax added.
  - Privileges can be given to a group using the "GROUP" key word.
 

For example:

```
GRANT SELECT ON foobar TO GROUP my_group;
```

The key word 'PUBLIC' is also supported to mean all users.

Privileges can only be granted or revoked to one user or group at a time.

"WITH GRANT OPTION" is not supported. Only class owners can change access control

- The default access control is to to grant users readonly access. You must explicitly grant insert/update access to users. To change this, modify the line in
 

```
src/backend/utils/acl.h
```

 that defines ACL\_WORLD\_DEFAULT

#### Bug fixes:

- \* the bug where aggregates of empty tables were not run has been fixed. Now, aggregates run on empty tables will return the initial conditions of the aggregates. Thus, COUNT of an empty table will now properly return 0. MAX/MIN of an empty table will return a row of value NULL.
- \* allow the use of \; inside the monitor
- \* the LISTEN/NOTIFY asynchronous notification mechanism now work
- \* NOTIFY in rule action bodies now work
- \* hash indexes work, and access methods in general should perform better. creation of large btree indexes should be much faster. (thanks to Paul Aoki)

#### Other changes and enhancements:

- \* addition of an EXPLAIN statement used for explaining the query execution plan (eg. "EXPLAIN SELECT \* FROM EMP" prints out the execution plan for the query).
- \* WARN and NOTICE messages no longer have timestamps on them. To turn on timestamps of error messages, uncomment the line in
 

```
src/backend/utils/elog.h:
```

```
/* define ELOG_TIMESTAMPS */
```
- \* On an access control violation, the message
 

```
"Either no such class or insufficient privilege"
```

 will be given. This is the same message that is returned when a class is not found. This dissuades non-privileged users from guessing the existence of privileged classes.
- \* some additional system catalog changes have been made that are not visible to the user.

#### libpgtcl changes:

- \* The -oid option has been added to the "pg\_result" tcl command. pg\_result -oid returns oid of the last row inserted. If the last command was not an INSERT, then pg\_result -oid returns "".
- \* the large object interface is available as pg\_lo\* tcl commands: pg\_lo\_open, pg\_lo\_close, pg\_lo\_creat, etc.

#### Portability enhancements and New Ports:

- \* flex/lex problems have been cleared up. Now, you should be able to use flex instead of lex on any platforms. We no longer make assumptions of what lexer you use based on the platform you use.
- \* The Linux-ELF port is now supported. Various configuration have been tested: The following configuration is known to work:
  - kernel 1.2.10, gcc 2.6.3, libc 4.7.2, flex 2.5.2, bison 1.24
 with everything in ELF format,

New utilities:

- \* ipcclean added to the distribution
  - ipcclean usually does not need to be run, but if your backend crashes and leaves shared memory segments hanging around, ipcclean will clean them up for you.

New documentation:

- \* the user manual has been revised and libpq documentation added.

## E.53. Postgres95 Release 0.02

**Release date:** 1995-05-25

### E.53.1. Modificações

Incompatible changes:

- \* The SQL statement for creating a database is 'CREATE DATABASE' instead of 'CREATEDB'. Similarly, dropping a database is 'DROP DATABASE' instead of 'DESTROYDB'. However, the names of the executables 'createdb' and 'destroydb' remain the same.

New tools:

- \* pgperl - a Perl (4.036) interface to Postgres95
- \* pg\_dump - a utility for dumping out a postgres database into a script file containing query commands. The script files are in a ASCII format and can be used to reconstruct the database, even on other machines and other architectures. (Also good for converting a Postgres 4.2 database to Postgres95 database.)

The following ports have been incorporated into postgres95-beta-0.02:

- \* the NetBSD port by Alistair Crooks
- \* the AIX port by Mike Tung
- \* the Windows NT port by Jon Forrest (more stuff but not done yet)
- \* the Linux ELF port by Brian Gallew

The following bugs have been fixed in postgres95-beta-0.02:

- \* new lines not escaped in COPY OUT and problem with COPY OUT when first attribute is a '.'
- \* cannot type return to use the default user id in createuser
- \* SELECT DISTINCT on big tables crashes
- \* Linux installation problems
- \* monitor doesn't allow use of 'localhost' as PGHOST
- \* psql core dumps when doing \c or \l
- \* the "pgtclsh" target missing from src/bin/pgtclsh/Makefile
- \* libpgtcl has a hard-wired default port number
- \* SELECT DISTINCT INTO TABLE hangs
- \* CREATE TYPE doesn't accept 'variable' as the internallength
- \* wrong result using more than 1 aggregate in a SELECT

## E.54. Postgres95 Release 0.01

**Release date:** 1995-05-01

Initial release.



# Apêndice F. O repositório CVS

Marc Fournier, Tom Lane, e Thomas Lockhart 1999-05-20

O código fonte do PostgreSQL é armazenado e gerenciado utilizando o sistema de gerenciamento de código CVS.<sup>1</sup>

Estão disponíveis pelo menos dois métodos, CVS anônimo e CVSup, para trazer a árvore de código CVS do servidor PostgreSQL para a máquina local.

## F.1. Obtenção do código fonte via CVS anônimo

Se for desejado se manter atualizado regularmente com os fontes correntes, pode-se trazê-los do servidor CVS e depois utilizar o CVS para realizar atualizações periódicas.

### CVS anônimo

1. É necessária uma cópia local do CVS, que pode ser obtida em CVS - Concurrent Versions System (<http://www.nongnu.org/cvs/>) (o sítio oficial com a última versão), ou em algum sítio de espelho. É recomendada a versão 1.10 ou mais nova. Muitos sistemas operacionais possuem uma versão recente do cvs instalada por padrão.

2. Efetuar o login inicial no servidor CVS:

```
cvs -d :pserver:anoncvs@anoncvs.postgresql.org:/projects/cvsroot login
```

Será solicitada uma senha; pode ser qualquer coisa, *exceto uma cadeia de caracteres vazia* (Portanto, não pressione a tecla Enter antes de digitar algum caractere - N. do T.).

Só é necessário realizar esta operação uma única vez, porque a senha é salva no arquivo `.cvspass` no diretório pessoal (home).

3. Trazer os fontes do PostgreSQL:

```
cvs -z3 -d :pserver:anoncvs@anoncvs.postgresql.org:/projects/cvsroot co -P pgsql
```

(`co` é abreviatura de `checkout`. N. do T.) Este comando coloca os fontes do PostgreSQL no subdiretório `pgsql` do diretório corrente.

**Nota:** Se for usado um acesso de banda larga à Internet, pode não haver necessidade da opção `-z3`, que instrui o CVS a usar compressão `gzip` na transferência dos dados, mas em um acesso na velocidade de modem há um ganho substancial.

A transferência inicial é um pouco mais lenta que simplesmente baixar o arquivo `tar.gz`; estima-se que demore 40 minutos ou mais com um modem de 28.8K. A vantagem do CVS não é vista até o momento em que se deseja atualizar o conjunto de arquivos.

4. Sempre que for desejado fazer uma atualização para obter os fontes mais recentes no CVS, o subdiretório `pgsql` deve ser tornado o diretório corrente (`cd`), e executado

```
$ cvs -z3 update -d -P
```

Este procedimento traz somente as alterações realizadas desde a última atualização. Normalmente a atualização demora somente poucos minutos, mesmo em um acesso na velocidade de modem.

5. É possível reduzir um pouco a digitação criando o arquivo `.cvsrc` no diretório pessoal contendo:

```
cvs -z3
update -d -P
```

Este arquivo define a opção global `-z3` para todos os comandos cvs, e as opções `-d` e `-P` para atualizações no cvs. Em seguida, basta executar

```
$ cvs update
```

para atualizar os arquivos.

## Cuidado

Algumas versões mais antigas do CVS possuem um erro que faz com que todos os arquivos trazidos sejam armazenados no diretório podendo ser escrito por todos. Caso isto aconteça, é possível fazer algo como

```
$ chmod -R go-w pgsql
```

para definir as permissões de forma apropriada. Este erro foi corrigido no CVS versão 1.9.28.

O CVS pode fazer muitas outras coisas, como trazer versões anteriores dos fontes do PostgreSQL em vez de trazer a última versão de desenvolvimento. Para obter informações adicionais deve ser consultado o manual que acompanha o CVS, ou deve ser consultado o manual em Version Management with CVS (<http://ximbiot.com/cvs/manual/>).

## F.2. Organização da árvore do CVS

**Autor:** Escrito por Marc G. Fournier (<[scrappy@hub.org](mailto:scrappy@hub.org)>) em 1998-11-05

O comando `cvs checkout` possui um sinalizador, `-r`, que permite trazer (`checkout`) uma determinada revisão de um módulo. Este sinalizador torna fácil, por exemplo, trazer os fontes que compõem a versão 6\_4 do módulo `tc` a qualquer momento:

```
$ cvs checkout -r REL6_4 tc
```

Este comando é útil, por exemplo, caso alguém diga que existe um erro nesta versão, mas o erro não pode ser encontrado na cópia de trabalho corrente.

**Dica:** Também é possível trazer um módulo, como este se encontrava em uma determinada data, utilizando a opção `-D`.

Quando se marca mais de um arquivo com a mesma marca, pode-se pensar na marca (`tag`) como sendo “uma curva traçada através de uma matriz de nome de arquivo versus número de revisão”.<sup>2</sup> Digamos que existam 5 arquivos com as seguintes revisões:

arq1	arq2	arq3	arq4	arq5	
1.1	1.1	1.1	1.1	1.1	1.1
1.2*-	1.2	1.2	-1.2*-		
1.3	\- 1.3*-	1.3	/ 1.3		
1.4		\ 1.4	/ 1.4		
		\-1.5*-	1.5		
		1.6			

então a marca `MARCA` faz referência a `arq1-1.2`, `arq2-1.3`, etc. (continuação tirada do manual do CVS->) Pode-se imaginar a marca como uma alça presa à curva traçada através das revisões marcadas. Quando se puxa a alça, são trazidas todas as revisões marcadas. Outra forma de enxergar é como sendo uma “visão” através de um conjunto de revisões que é um “plano” ao longo das revisões marcadas, como mostrado abaixo:

arq1	arq2	arq3	arq4	arq5	
		1.1			
		1.2			
	1.1	1.3			
1.1	1.2	1.4	1.1		
1.2*----	1.3*----	1.5*----	1.2*----	1.1	(--- <--- Olhe por aqui
1.3		1.6	1.3		\_
1.4			1.4		
			1.5		

(<-fim da continuação do manual - N. do T.)

**Nota:** É o mesmo que criar uma ramificação (`branch`) de versão, sem utilizar a opção `-b` adicionada ao comando.

Portanto, para criar a versão 6.4 foi feito

```
$ cd postgresql
$ cvs tag -b REL6_4
```

para criar a marca e a ramificação na árvore RELEASE.

Para os que possuem acesso ao CVS, é simples criar diretórios para versões diferentes. Primeiro devem ser criados dois subdiretórios, RELEASE e CURRENT, para não confundir as duas coisas. Depois se executa:

```
cd RELEASE
cvs checkout -P -r REL6_4 postgresql
cd ../CURRENT
cvs checkout -P postgresql
```

que resulta em duas árvores de diretório, RELEASE/postgresql e CURRENT/postgresql. Deste ponto em diante, o CVS acompanha qual ramificação do repositório está em qual árvore de diretório, permitindo atualizações independentes para cada árvore.<sup>3</sup>

Para trabalhar *apenas* na árvore de fontes CURRENT, basta fazer tudo como estava sendo feito antes de começar a colocar marcas de ramificação de versão.

Após a baixa (checkout) inicial de uma ramificação

```
$ cvs checkout -r REL6_4
```

tudo que é feito dentro desta estrutura de diretório fica restrito a esta ramificação. Se for aplicada uma correção a esta estrutura de diretório e feito um

```
cvs commit
```

de dentro desta árvore, a correção é aplicada a esta ramificação, e *somente* a esta ramificação.<sup>4</sup>

## F.3. Obtenção do código fonte via CVSup

O CVSup é uma alternativa ao uso do CVS anônimo para trazer a árvore de fontes do PostgreSQL. O CVSup foi desenvolvido por John Polstra (<jdp@polstra.com>) para distribuir repositórios CVS e outras árvores de arquivos para o projeto FreeBSD (<http://www.freebsd.org>).

Uma das principais vantagens em utilizar o CVSup é que este pode replicar, confiavelmente, *todo* o repositório CVS no sistema local, permitindo acesso local rápido às operações do CVS, como `log` e `diff`. Entre outras vantagens está a sincronização rápida com o servidor PostgreSQL devido a um protocolo de transferência de fluxo eficiente, que envia apenas as modificações realizadas desde a última atualização.

### F.3.1. Preparação do sistema cliente do CVSup

São necessárias duas áreas de diretório para o CVSup realizar seu trabalho: um repositório CVS local (ou simplesmente uma área de diretório, se estiver sendo trazido um instantâneo em vez do repositório; veja abaixo), e uma área local para registrar as transações do CVSup. Estas duas áreas podem coexistir na mesma árvore de diretório.

Decidir onde se deseja manter a cópia local do repositório CVS. Em um dos nossos sistemas foi definido recentemente o repositório em `/home/cvs/`, mas estava sendo mantido anteriormente sob a árvore de desenvolvimento do PostgreSQL em `/opt/postgres/cvs/`. Caso se pretenda manter o repositório em `/home/cvs/`, então deve ser colocado

```
setenv CVSROOT /home/cvs
```

no arquivo `.cshrc` do usuário, ou uma linha semelhante no arquivo `.bashrc` ou `.profile`, dependendo do interpretador de comandos utilizado.

A área de repositório do cvs deve ser inicializada. Após CVSROOT ser definido, então a inicialização pode ser feita com um único comando:

```
$ cvs init
```

após o qual deve ser visto pelo menos um diretório chamado CVSROOT ao se listar o diretório CVSROOT:

```
$ ls $CVSROOT
CVSROOT/
```

### F.3.2. Execução do cliente do CVSup

Verificar se o programa cvsup está no caminho de procura; na maioria dos sistemas operacionais é feito digitando

```
which cvsup
```

Depois, basta executar o cvsup utilizando:

```
$ cvsup -L 2 postgres.cvsup
```

onde -L 2 habilita algumas mensagens de status, permitindo monitorar a evolução da atualização, e *postgres.cvsup* é o caminho e nome atribuído ao arquivo de configuração do CVSup.

Abaixo está mostrado um arquivo de configuração do CVSup modificado para uma instalação específica, e que mantém um repositório local CVS completo:

```
# This file represents the standard CVSup distribution file
# for the PostgreSQL ORDBMS project
# Modified by lockhart@fourpalms.org 1997-08-28
# - Point to my local snapshot source tree
# - Pull the full CVS repository, not just the latest snapshot
#
# Defaults that apply to all the collections
*default host=cvsup.postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
# enable the following line to get the latest snapshot
#*default tag=.
# enable the following line to get whatever was specified above or by default
# at the date specified below
#*default date=97.08.29.00.00.00

# base directory where CVSup will store its 'bookmarks' file(s)
# will create subdirectory sup/
#*default base=/opt/postgres # /usr/local/pgsql
*default base=/home/cvs

# prefix directory where CVSup will store the actual distribution(s)
*default prefix=/home/cvs

# complete distribution, including all below
pgsql

# individual distributions vs 'the whole thing'
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

Na configuração acima, Se for especificada a opção *repository* em vez de *pgsql*, será obtida uma cópia completa de todo o repositório localizado em *cvsup.postgresql.org*, incluindo o diretório CVSROOT. Se isto for feito, provavelmente será desejado excluir os arquivos neste diretório que se desejar modificar localmente, utilizando um arquivo de recusa. Por exemplo, para a configuração acima pode ser criado o arquivo */home/cvs/sup/repository/refuse*:

```
CVSROOT/config*
CVSROOT/commitinfo*
CVSROOT/logininfo*
```

Para se informar sobre como usar os arquivos de recusa, devem ser consultadas as páginas do manual do CVSup.

A seguir está mostrado o arquivo de configuração sugerido do CVSup, tirado do site de ftp do PostgreSQL (<ftp://ftp.postgresql.org/pub/CVSup/README.cvsup>), que traz o instantâneo corrente apenas:

```
# This file represents the standard CVSup distribution file
# for the PostgreSQL ORDBMS project
#
# Defaults that apply to all the collections
*default host=cvsup.postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
*default tag=.

# base directory where CVSup will store its 'bookmarks' file(s)
*default base=/usr/local/pgsql

# prefix directory where CVSup will store the actual distribution(s)
*default prefix=/usr/local/pgsql

# complete distribution, including all below
pgsql

# individual distributions vs 'the whole thing'
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

### F.3.3. Instalação do CVSup

O CVSup está disponível sob a forma de fonte, binários pré-construídos, e RPMs do Linux. É muito mais fácil usar os binários que construir a partir do fonte, principalmente porque é necessário para a construção o muito poderoso, mas muito grande, compilador Modula-3.

#### Instalação do CVSup a partir dos binários

Podem ser utilizados binários pré-construídos se for utilizada uma plataforma para a qual os binários estão disponíveis no site de ftp do PostgreSQL (<ftp://ftp.postgresql.org/pub>), ou se estiver utilizando o FreeBSD, para o qual o CVSup está disponível como um “port”.<sup>5</sup>

**Nota:** O CVSup foi desenvolvido originalmente como uma ferramenta para distribuição da árvore de fontes do FreeBSD. Está disponível como um “port” e, para os usuários do FreeBSD, se não for suficiente informar como obter e instalá-lo então, por favor, contribua enviando o procedimento.

Quando esta documentação foi escrita, haviam binários disponíveis para Alpha/Tru64, ix86/xBSD, HPPA/HP-UX 10.20, MIPS/IRIX, ix86/linux-libc5, ix86/linux-glibc, Sparc/Solaris e Sparc/SunOS.

1. Obter o arquivo binário tar do cvsup ( para ser um cliente não é necessário cvsud) apropriado para a plataforma sendo usada.
  - a. Se estiver sendo utilizado o FreeBSD, deve ser instalado o “port” do CVSup.
  - b. Se estiver sendo utilizada uma outra plataforma, deve ser procurado e baixado o binário apropriado no sítio de FTP do PostgreSQL (<ftp://ftp.postgresql.org/pub>).
2. Analisar o arquivo tar para verificar o conteúdo e a estrutura de diretório, se houver alguma. Pelo menos no arquivo tar do Linux, o binário estático e as páginas do manual estão incluídas no pacote sem nenhum diretório.
  - a. Se o binário estiver no nível mais alto do arquivo tar, então simplesmente deve ser feita a extração do arquivo tar no diretório de destino:
 

```
$ cd /usr/local/bin
```

```
$ tar zxvf /usr/local/src/cvsup-16.0-linux-i386.tar.gz
$ mv cvsup.1 ../doc/man/man1/
```

- b. Havendo uma estrutura de diretório no arquivo tar, então o arquivo tar deve ser extraído dentro do diretório /usr/local/src, e os binários movidos para o local apropriado conforme mostrado acima.

3. Deve-se garantir que os novos binários podem ser encontrados no caminho de procura.

```
$ rehash
$ which cvsup
$ set path=(caminho para o cvsup $path)
$ which cvsup
/usr/local/bin/cvsup
```

### F.3.4. Instalação a partir dos fontes

A instalação do CVSup a partir dos arquivos fonte não é totalmente trivial, principalmente porque na maioria dos sistemas é necessário instalar antes o compilador Modula-3. Este compilador está disponível como um arquivo RPM para o Linux, um pacote FreeBSD, ou como código fonte.

**Nota:** A instalação do Modula-3 a partir dos fontes ocupa cerca de 200MB de espaço em disco, que fica reduzido para cerca de 50MB de espaço quando os fontes são removidos.

#### Instalação no Linux

1. Instalar o Modula-3.
  - a. Obter a distribuição do Modula-3 em Polytechnique Montréal Modula-3 (PM3) (<http://www.elegosoft.com/pm3/>), que está ativamente mantendo o código base desenvolvido originalmente pelo DEC Systems Research Center (<http://www.research.digital.com/SRC/modula-3/html/home.html>). A distribuição do RPM do PM3 comprimida tem aproximadamente 30MB. Quando este documento foi escrito, a versão 1.1.10-1 instalava sem problemas no RH-5.2, enquanto a versão 1.1.11-1 era aparentemente feita para outra versão (RH-6.0?), e não executava no RH-5.2.

**Dica:** Este empacotamento de rpm em particular possui *muitos* arquivos RPM e, portanto, provavelmente se desejará colocá-los em um diretório separado.

- b. Instalar os rpms do Modula-3:

```
# rpm -Uvh pm3*.rpm
```

2. Descompactar a distribuição do cvsup:
 

```
# cd /usr/local/src
# tar zxf cvsup-16.0.tar.gz
```
3. Construir a distribuição do cvsup, suprimindo a funcionalidade de interface gráfica para evitar a necessidade de bibliotecas do X11:
 

```
# make M3FLAGS="-DNOGUI"
```

e, se for desejado construir um binário estático que possa ser movido para sistemas que não possuam o Modula-3 instalado, tente:

```
# make M3FLAGS="-DNOGUI -DSTATIC"
```
4. Instalar o binário construído:
 

```
# make M3FLAGS="-DNOGUI -DSTATIC" install
```

## Notas

1. Pode ser encontrado um bom tutorial sobre o CVS em EAD/CCUEC Mini Cursos Virtuais - CVS ([http://www.ead.unicamp.br/minicurso/cvs/texto/tabela\\_conteudo.html](http://www.ead.unicamp.br/minicurso/cvs/texto/tabela_conteudo.html)). (N. do T.)
2. Número de revisão — Cada versão do arquivo possui um número de revisão único. Os números de revisão se parecem com “1.1”, “1.2”, “1.3.2.2”, ou mesmo “1.3.2.2.4.5”. Um número de revisão sempre possui um número uniforme de inteiros decimais separados por ponto. Por padrão, a revisão “1.1” é a primeira revisão do arquivo. A cada revisão

sucessiva é atribuído um novo número acrescentando um ao número mais à direita. Também é possível haver números contendo mais de um ponto como, por exemplo, “1.3.2.2”. Estas revisões representam revisões em ramificações. CVS--Concurrent Versions System v1.12.12.1: Revisions ([http://ximbiot.com/cvs/wiki/index.php?title=CVS--Concurrent\\_Versions\\_System\\_v1.12.12.1:\\_Revisions](http://ximbiot.com/cvs/wiki/index.php?title=CVS--Concurrent_Versions_System_v1.12.12.1:_Revisions)) (N. do T.)

3. Se for desejado trazer duas ramificações como, por exemplo, REL6\_4 e REL7\_4\_1, é necessário criar os subdiretórios RELEASE/REL6\_4 e RELEASE/REL7\_4\_1, e executar os comandos “cvs checkout -P -r REL6\_4 pgsql” e “cvs checkout -P -r REL7\_4\_1 pgsql” a partir destes subdiretórios. (N. do T.)
4. Dentro de cada árvore existe o arquivo pgsql/CVS/Tag contendo o nome da ramificação. (N. do T.)
5. Para o Debian basta executar “apt-get install cvsup”. (N. do T.)

# Apêndice G. Documentação

O PostgreSQL possui quatro formatos de documentação principais:

- Texto puro, para informações de pré-instalação.
- HTML, para navegação on-line e referência.
- PDF ou Postscript, para impressão.
- `man pages` para referência rápida.<sup>1</sup>

Além desses, podem ser encontrados vários arquivos texto puro README dentro da árvore do código fonte do PostgreSQL, documentando vários tópicos da implementação.

A documentação em HTML e as `man pages` fazem parte da distribuição padrão, sendo instaladas por padrão. Os formatos PDF e Postscript da documentação estão disponíveis em separado, podendo ser baixados.

## G.1. DocBook

Os fontes da documentação estão escritos em *DocBook*, que é uma linguagem de marcação semelhante, superficialmente, ao HTML. Estas duas linguagens são aplicações da *Linguagem Padrão de Marcação Generalizada* (Standard Generalized Markup Language), SGML, que é essencialmente uma linguagem para descrever outras linguagens. No que vem a seguir, é utilizado tanto o termo DocBook quanto SGML, mas tecnicamente estes dois termos não são intercambiáveis.

O DocBook permite ao autor especificar a estrutura e o conteúdo de um documento técnico, sem se preocupar com os detalhes da apresentação. O estilo do documento define como o conteúdo é apresentado em uma dentre várias formas finais. O DocBook é mantido pelo grupo OASIS (<http://www.oasis-open.org>). O site oficial do DocBook (<http://www.oasis-open.org/docbook>) possui uma boa documentação introdutória e de referência, além de um livro completo da O'Reilly que pode ser lido on-line. O Projeto de Documentação do FreeBSD (<http://www.freebsd.org/docproj/docproj.html>) também usa o DocBook e possui boas informações, incluindo vários guias para estilo que valem a pena serem levados em consideração.

## G.2. Conjunto de ferramentas

As seguintes ferramentas são utilizadas para processar a documentação. Algumas podem ser opcionais, conforme indicado.

DTD do DocBook (<http://www.oasis-open.org/docbook/sgml/>)

Esta é a definição do próprio DocBook. Atualmente é utilizada a versão 4.2; não pode ser usada uma versão anterior ou posterior. Deve ser observado que também existe uma versão XML do DocBook — não utilize esta versão.

Entidades de caractere ISO 8879 (<http://www.oasis-open.org/cover/ISOEnts.zip>)

Estas entidades são requeridas pelo DocBook, mas são distribuídas em separado porque são mantidas pela ISO.

OpenJade (<http://openjade.sourceforge.net>)

Este é o pacote básico para processar o SGML. Contém um analisador SGML, um processador DSSSL (ou seja, um programa para converter SGML em outros formatos utilizando as folhas de estilo DSSSL), assim como várias ferramentas relacionadas. O Jade agora é mantido pelo grupo OpenJade, e não mais por James Clark.

Folhas de estilo DSSSL do DocBook (<http://docbook.sourceforge.net/projects/dsssl/index.html>)

Contém instruções de processamento para converter os fontes em DocBook para outros formatos, tal como HTML.

Ferramentas DocBook2X (<http://docbook2x.sourceforge.net>)

Este pacote opcional é utilizado para criar as `man pages`, e contém vários pacotes de pré-requisitos próprios. Verifique o sítio na Web.

JadeTeX (<http://jadetex.sourceforge.net>)

Se for desejado, também pode ser instalado o JadeTeX para utilizar o TeX como formatador para o Jade. O JadeTeX pode criar arquivos Postscript ou PDF (este último com marcadores).



Entretanto, a saída do JadeTeX é inferior à obtida pelo processador do RTF. Entre as áreas com problemas específicos estão as tabelas e vários artefatos de espaçamento vertical e horizontal. Além disso, não existe oportunidade para melhorar manualmente os resultados.

Existem experiências documentadas sobre vários métodos de instalação para as várias ferramentas necessárias para processar a documentação, as quais estão descritas abaixo. Podem haver outros pacotes contendo distribuições destas ferramentas. Por favor informe o status do pacote para a lista de discussão da documentação, e esta informação será incluída aqui.

### G.2.1. Instalação de RPM no Linux

A maioria dos fornecedores disponibiliza um conjunto completo de pacotes RPM para processar o DocBook em suas distribuições. Procure pela opção “SGML” ao instalar, ou pelos seguintes pacotes: `sgml-common`, `docbook`, `stylesheets`, `openjade` (ou `jade`). É possível que também seja necessário o pacote `sgml-tools`. Se estes pacotes não estiverem disponíveis na distribuição utilizada, então deve ser possível utilizar pacotes de alguma outra distribuição, razoavelmente compatível.

### G.2.2. Instalação no FreeBSD

O próprio “Projeto de Documentação do FreeBSD” é um grande usuário do DocBook e, portanto, não é surpresa que exista um conjunto completo das ferramentas de documentação “portadas” disponível no FreeBSD. É necessário instalar as seguintes ferramentas para gerar a documentação no FreeBSD.

- `textproc/sp`
- `textproc/openjade`
- `textproc/iso8879`
- `textproc/dsssl-docbook-modular`

Aparentemente, não existe um DocBook V4.2 SGML DTD portado disponível atualmente, sendo necessário efetuar sua instalação manualmente.

Também podem ser de interesse vários outros aplicativos contidos em `/usr/ports/print` (`tex`, `jadetex`).

É possível que os aplicativos portados não atualizem o arquivo do catálogo principal em `/usr/local/share/sgml/catalog`. Certifique-se que a seguinte linha está presente:

```
CATALOG "/usr/local/share/sgml/docbook/4.2/docbook.cat"
```

Se não for desejado editar o arquivo, também pode ser definida a variável de ambiente `SGML_CATALOG_FILES` como uma lista separada por vírgulas de arquivos de catálogo (tal como mostrado acima).

Podem ser encontradas informações adicionais sobre as ferramentas de documentação do FreeBSD nas Instruções do Projeto de Documentação do FreeBSD ([http://www.freebsd.org/doc/en\\_US.ISO8859-1/books/fdp-primer/tools.html](http://www.freebsd.org/doc/en_US.ISO8859-1/books/fdp-primer/tools.html)).

### G.2.3. Pacotes do Debian

Existe um conjunto completo de pacotes para as ferramentas de documentação disponível para o Debian GNU/Linux. Para instalar, deve simplesmente ser utilizado: <sup>2</sup>

```
apt-get install jade
apt-get install docbook
apt-get install docbook-stylesheets
```

### G.2.4. Instalação manual a partir dos fontes

O processo de instalação manual das ferramentas do DocBook é um tanto complexo e, portanto, havendo pacotes pré-construídos disponíveis estes devem ser utilizados. Aqui é descrita apenas a instalação padrão, com caminhos de instalação padrão razoáveis, e nenhuma funcionalidade “extravagante”. Para obter detalhes, deve ser estudada a documentação do próprio pacote, e lido o material introdutório do SGML.

### G.2.4.1. Instalação do OpenJade

1. A instalação do OpenJade possui um processo de construção `./configure; make; make install` ao estilo GNU. Os detalhes podem ser encontrados na distribuição do código fonte do OpenJade. De forma concisa:

```
./configure --enable-default-catalog=/usr/local/share/sgml/catalog
make
make install
```

Não se esqueça onde foi colocado o “catálogo padrão”, pois será necessário abaixo. Também pode ser deixado de fora, mas deverá ser definida a variável de ambiente `SGML_CATALOG_FILES` para apontar para o arquivo sempre que o jade for utilizado posteriormente (Este método também é uma opção se OpenJade já estiver instalado, e for desejado instalar o restante do conjunto de ferramentas localmente).

2. Além disso, devem ser instalados os arquivos `dsssl.dtd`, `fot.dtd`, `style-sheet.dtd` e `catalog` do diretório `dsssl` em algum lugar, talvez no diretório `/usr/local/share/sgml/dsssl`. Provavelmente é mais fácil copiar todo o diretório:

```
cp -R dsssl /usr/local/share/sgml
```

3. Para terminar, deve ser criado o arquivo `/usr/local/share/sgml/catalog` e adicionada a seguinte linha ao mesmo:

```
CATALOG "dsssl/catalog"
```

(Esta é uma referência ao caminho relativo do arquivo instalado no passo 2. Assegure de ajustá-la se for escolhido um posicionamento diferente para a instalação).

### G.2.4.2. Instalação do kit DTD do DocBook

1. Obter a distribuição DocBook V4.2 (<http://www.docbook.org/sgml/4.2/docbook-4.2.zip>).
2. Criar o diretório `/usr/local/share/sgml/docbook-4.2` e torná-lo o diretório corrente (O local exato é irrelevante, mas este é razoável dentro do posicionamento que está sendo seguido).

```
$ mkdir /usr/local/share/sgml/docbook-4.2
$ cd /usr/local/share/sgml/docbook-4.2
```

3. Descompactar o arquivo.

```
$ unzip -a ...../docbook-4.2.zip
```

(Os arquivos serão descompactados no diretório corrente)

4. Editar o arquivo `/usr/local/share/sgml/catalog` (ou o que foi informado ao jade durante a instalação) e adicionar uma linha como esta:

```
CATALOG "docbook-4.2/docbook.cat"
```

5. Obter o arquivo Entidades caractere ISO 8879 (<http://www.oasis-open.org/cover/ISOEnts.zip>), e descompactá-lo colocando os arquivos no mesmo diretório onde foram colocados os arquivos do DocBook.

```
$ cd /usr/local/share/sgml/docbook-4.2
$ unzip ...../ISOEnts.zip
```

6. Executar o seguinte comando no diretório contendo os arquivos do DocBook e do ISO:

```
perl -pi -e 's/iso-(.*).gml/ISO\1/g' docbook.cat
```

(Este procedimento corrige uma discrepância entre os nomes utilizados no arquivo de catálogo do DocBook e os nomes verdadeiros dos arquivos das entidades caractere ISO).

### G.2.4.3. Instalação das folhas de estilo DSSSL do DocBook

Para instalar as folhas de estilo os arquivos deve ser descompactados e movidos para um lugar adequado como, por exemplo, `/usr/local/share/sgml` (O arquivo compactado cria automaticamente um subdiretório ao ser descompactado).

```
$ gunzip docbook-dsssl-1.xx.tar.gz
$ tar -C /usr/local/share/sgml -xf docbook-dsssl-1.xx.tar
```

Também pode ser colocada a entrada usual do catálogo em `/usr/local/share/sgml/catalog`:

```
CATALOG "docbook-dsssl-1.xx/catalog
```

Como as folhas de estilo mudam muito freqüentemente, sendo alguma vezes benéfico tentar versões alternativas, o PostgreSQL não utiliza esta entrada do catálogo. Consulte a Seção G.2.5 para obter informações sobre como selecionar as folhas de estilo.

#### G.2.4.4. Instalação do JadeTeX

Para instalar e utilizar o JadeTeX, é necessário haver uma instalação do TeX e do LaTeX2<sub>ε</sub> funcionando, incluindo os pacotes suportados `tools` e `graphics`, Babel, fontes AMS e AMS-LaTeX, a extensão PSNFSS e o kit que acompanha “as 35 fontes”, o programa `dvips` para gerar PostScript, os pacotes de macros `fancyhdr`, `hyperref`, `minitoc`, `url` e `ot2enc`. Todos estes podem ser encontrados CTAN (<http://www.ctan.org>). A instalação do sistema básico do TeX está muito acima do escopo desta introdução. Devem existir pacotes binários disponíveis para qualquer sistema que possa executar o TeX.

Antes de ser possível utilizar o JadeTeX com os fontes da documentação do PostgreSQL, é necessário aumentar o tamanho das estruturas de dado internas do TeX. Os detalhes podem ser encontrados nas instruções de instalação do JadeTeX.

Após terminar esta parte pode ser instalado o JadeTeX:

```
$ gunzip jadetex-xxx.tar.gz
$ tar xf jadetex-xxx.tar
$ cd jadetex
$ make install
$ mktexlsr
```

Os dois últimos comando devem ser executados como root.

#### G.2.5. Detecção pelo configure

Para se poder construir a documentação, primeiro é necessário executar o script `configure`, como é feito para gerar os próprios programas do PostgreSQL. Deve ser verificado se a saída perto do fim da execução se parece com:

```
checking for onsgmls... onsgmls
checking for openjade... openjade
checking for DocBook V4.2... yes
checking for DocBook stylesheets... /usr/lib/sgml/stylesheets/nwalsh-modular
checking for sgmlspl... sgmlspl
```

Se não for encontrado nem o `onsgmls` nem o `nsgmls`, então as outras quatro linhas não serão vistas. O `nsgmls` é parte do pacote Jade. Se não for encontrado o “DocBook V4.2”, então o kit `DocBook DTD` não foi instalado em um local onde o jade possa encontrá-lo, ou os arquivos do catálogo não foram definidos corretamente. Devem ser vistas as dicas de instalação acima. As folhas de estilo do DocBook são procuradas em vários locais relativamente padrão, mas se estiverem em algum outro local deve ser definida a variável de ambiente `DOCBOOKSTYLE` com o local correto, e executado novamente `configure` após isto.

### G.3. Geração da documentação

Após estar tudo instalado, o diretório `doc/src/sgml` deve ser tornado o diretório corrente, e um dos comandos descritos nas subseções abaixo deve ser executado para gerar a documentação (Lembre-se de utilizar o `make` do GNU).

#### G.3.1. HTML

Para gerar a versão HTML da documentação:

```
doc/src/sgml$ gmake html
```

Esta também é a versão gerada por padrão.

Quando a documentação HTML é gerada, o processamento também gera as informações de vínculo com as entradas do índice. Portanto, se for desejado que a documentação possua um índice no final, é necessário gerar a documentação HTML primeiro, e depois gerar a documentação novamente no formato desejado.

Para permitir o tratamento mais fácil da distribuição final, os arquivos que compõem a documentação HTML são armazenados em um arquivo `tar` que é desempacotado durante a instalação. Para criar o pacote de documentação HTML, devem ser utilizados os comandos

```
cd doc/src
gmake postgres.tar.gz
```

Na distribuição, estes arquivos se encontram no diretório `doc`, sendo instalados por padrão pelo `gmake install`.

### G.3.2. Páginas do manual Unix

É utilizado o utilitário `docbook2man` para converter as páginas `refentry` do DocBook em saída `*roff` adequada para as páginas do manual. As páginas do manual também são distribuídas como um arquivo `tar`, semelhante à versão HTML. Para gerar o pacote de páginas do manual devem ser utilizados os comandos

```
cd doc/src
gmake man.tar.gz
```

que produz um arquivo `tar` gerado no diretório `doc/src`.

Para gerar páginas do manual com qualidade, pode ser necessário utilizar uma versão “hackeada” do utilitário de conversão, ou fazer algum pós-processamento manual. Todas as páginas do manual devem ser inspecionadas manualmente antes de serem distribuídas.

### G.3.3. Imprimir a saída usando JadeTex

Se for desejado utilizar o JadeTex para gerar uma versão da documentação que possa ser impressa, pode ser utilizado um dos seguintes comandos:

- Para gerar a versão DVI:

```
doc/src/sgml$ gmake postgres.dvi
```

- Para gerar o Postscript a partir do DVI:

```
doc/src/sgml$ gmake postgres.ps
```

- Para gerar o PDF:

```
doc/src/sgml$ gmake postgres.pdf
```

(Obviamente pode ser gerada a versão PDF a partir do Postscript, mas se o PDF for gerado diretamente, terá hiperligações e outras funcionalidades avançadas).

### G.3.4. Imprimir a saída através do RTF

Também é possível criar uma versão imprimível da documentação do PostgreSQL convertendo-a em RTF, e depois aplicando pequenas correções de formatação utilizando um pacote de automação de escritórios. Dependendo das funcionalidades do pacote de automação de escritórios utilizado, a documentação pode ser convertida para Postscript a partir do PDF. O procedimento abaixo mostra este processo utilizando o Applixware.

**Nota:** Parece que a versão corrente da documentação do PostgreSQL dispara algum erro ou excede o limite de tamanho do OpenJade. Se o processo de geração da versão RTF demorar muito tempo, e o tamanho do arquivo de saída permanecer igual a 0, então você esbarrou no problema mas tenha em mente que a geração normal leva de 5 a 10 minutos, portanto não interrompa muito rapidamente.

#### Limpeza do RTF usando o Applixware

O OpenJade omite a especificação do estilo padrão para o corpo do texto. No passado, este problema não diagnosticado levava a um longo processo para geração do sumário. Entretanto, com grande ajuda das pessoas da Applixware, o sintoma foi diagnosticado e uma correção está disponível.

1. Gerar a versão RTF digitando:

```
doc/src/sgml$ gmake postgres.rtf
```

2. Reparar o arquivo RTF para especificar corretamente todos os estilos, em particular o estilo padrão. Se o documento contém seções `refentry`, também devem ser substituídas as dicas de formatação que ligam o parágrafo precedente ao parágrafo corrente e, em seu lugar, ligar o parágrafo corrente ao parágrafo seguinte. O utilitário `fixrtf` está disponível em `doc/src/sgml` para efetuar estes reparos:

```
doc/src/sgml$ ./fixrtf --refentry postgres.rtf
```

Este script adiciona `{\s0 Normal;}` como sendo o zero-ésimo estilo no documento. De acordo com a Applixware, o padrão RTF proíbe adicionar um zero-ésimo estilo implícito, embora o Word da Microsoft trate este caso. Para reparar as seções `refentry`, o script substitui as marcas `\keepn` por `\keep`.

3. Abra um novo documento no Applixware Words e depois importe o arquivo RTF.
4. Gere o novo Sumário (ToC) utilizando o Applixware.
  - a. Selecione as linhas existentes no Sumário, do início do primeiro caractere da primeira linha ao último caractere da última linha.
  - b. Construa um novo Sumário utilizando **Tools→Book Building→Create Table of Contents**. Selecione os três primeiros níveis de cabeçalho para serem incluídos no Sumário. Este procedimento substitui as linhas existentes importadas no RTF por um Sumário nativo do Applixware.
  - c. Ajuste a formatação do Sumário utilizando **Format→Style**, selecione cada um dos três estilos de Sumário, e ajuste o recuo para *First* e *Left*. Use os seguintes valores:

Style	First Indent (inches)	Left Indent (inches)
TOC-Heading 1	0.4	0.4
TOC-Heading 2	0.8	0.8
TOC-Heading 3	1.2	1.2

5. Percorra o documento para:
  - Ajustar as quebras de página.
  - Ajustar as larguras das colunas das tabelas.
6. Substitua os números das páginas alinhados à direita na parte de Exemplos e de Figuras do Sumário pelos valores corretos. Esta atividade só toma alguns minutos.
7. Apague a seção de índice do documento caso esteja vazia.
8. Gere novamente e ajuste o Sumário.
  - a. Selecione o campo Sumário.
  - b. Selecione **Tools→Book Building→Create Table of Contents**.
  - c. Desvincule o Sumário selecionando **Tools→Field Editing→Unprotect**.
  - d. Apague a primeira linha do Sumário, que é uma entrada para o próprio Sumário.
9. Salve o documento no formato nativo do Applixware Words para facilitar uma edição posterior, caso seja necessário.
10. Imprima ("Print") o documento em um arquivo no formato Postscript.

### G.3.5. Arquivos texto puro

Diversos arquivos são distribuídos como texto puro, para serem lidos durante o processo de instalação. O arquivo `INSTALL` corresponde ao Capítulo 14, com pequenas alterações para levar em conta a diferença de contexto. Para recriar este arquivo, o diretório `doc/src/sgml` deve ser tornado o diretório corrente, e deve ser executado **gmake INSTALL**. Isto cria o arquivo `INSTALL.html` que pode ser salvo como texto utilizando o Netscape Navigator e colocado no lugar do arquivo existente. O Netscape parece oferecer a melhor qualidade na conversão de HTML em texto (melhor que o lynx e o w3m).

O arquivo `HISTORY` pode ser criado de forma semelhante, utilizando o comando `gmake HISTORY`. Para criar o arquivo `src/test/regress/README` o comando é `gmake regress_README`.

### G.3.6. Verificação da sintaxe

A geração da documentação pode ser demorada. Entretanto, existe um método para verificar apenas se a sintaxe dos arquivos de documentação está correta que só leva alguns segundos:

```
doc/src/sgml$ gmake check
```

## G.4. Criação da documentação

O SGML e o DocBook não sofrem o mal do número excessivo de ferramentas de código aberto para criação de páginas. O conjunto de ferramentas mais comum é o editor Emacs/XEmacs com o modo apropriado de edição. Em alguns sistemas estas ferramentas fazem parte da instalação típica completa.

### G.4.1. Emacs/PSGML

O PSGML é o modo de editar documentos SGML mais comum e mais poderoso. Quando configurado de forma apropriada, permite utilizar o Emacs para inserir marcas e verificar a consistência da marcação. Também pode ser utilizado para HTML. Visite o página na Web do PSGML ([http://www.lysator.liu.se/projects/about\\_psgml.html](http://www.lysator.liu.se/projects/about_psgml.html)) para baixar os arquivos, obter as instruções de instalação, e ver a documentação detalhada.

Existe um ponto importante a ser observado com relação ao PSGML: o autor assume que o diretório principal da DTD do SGML na máquina utilizada é `/usr/local/lib/sgml`. Se for utilizado `/usr/local/share/sgml`, como acontece nos exemplos deste capítulo, isto deve ser compensado, seja definindo a variável de ambiente `SGML_CATALOG_FILES`, ou personalizando a instalação do PSGML (o manual explica como fazer).

O que vem a seguir deve ser adicionado ao arquivo de ambiente `~/ .emacs` (ajustando os nomes dos caminhos conforme apropriado para o sistema utilizado):

```
; ***** for SGML mode (psgml)

(setq sgml-omittag t)
(setq sgml-shorttag t)
(setq sgml-minimize-attributes nil)
(setq sgml-always-quote-attributes t)
(setq sgml-indent-step 1)
(setq sgml-indent-data t)
(setq sgml-parent-document nil)
(setq sgml-default-dtd-file "./reference.ced")
(setq sgml-exposed-tags nil)
(setq sgml-catalog-files '("/usr/local/share/sgml/catalog"))
(setq sgml-ecat-files nil)

(autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t )
```

e no mesmo arquivo deve ser adicionada uma entrada para SGML na definição (existente) de `auto-mode-alist`:

```
(setq
  auto-mode-alist
  '(("\\.sgml$" . sgml-mode)
  ))
```

Atualmente, cada arquivo fonte SGML possui o seguinte bloco no final:

```
<!-- Keep this comment at the end of the file
Local variables:
mode: sgml
sgml-omittag:t
sgml-shorttag:t
sgml-minimize-attributes:nil
sgml-always-quote-attributes:t
```

```
sgml-indent-step:1
sgml-indent-data:t
sgml-parent-document:nil
sgml-default-dtd-file:"./reference.ced"
sgml-exposed-tags:nil
sgml-local-catalogs:("/usr/lib/sgml/catalog")
sgml-local-ecat-files:nil
End:
-->
```

Isto define vários parâmetros do modo de edição, mesmo que o arquivo `~/ .emacs` não esteja definido mas, por azar, se forem seguidas as instruções de instalação acima, então o caminho do catálogo não corresponde a este local na máquina utilizada. Por isso pode ser necessário desabilitar as variáveis locais:

```
(setq inhibit-local-variables t)
```

A distribuição do PostgreSQL inclui o arquivo de definições DTD analisado `reference.ced`. Ao utilizar o PSGML, acaba-se descobrindo que uma maneira confortável de trabalhar com os arquivos separados das partes do livro é inserindo uma declaração `DOCTYPE` apropriada ao editar. Por exemplo, ao se trabalhar com o fonte deste arquivo, que é um capítulo do apêndice, o documento pode ser especificado como uma instância do “appendix” do documento DocBook fazendo a primeira linha se parecer com:

```
<!DOCTYPE appendix PUBLIC "-//OASIS//DTD DocBook V4.2//EN">
```

Isto permite que tudo que leia SGML o faça direito, e que o documento possa ser verificado pelo comando `nsgmls -s docguide.sgml`; mas esta linha precisa ser removida antes de ser gerado o conjunto completo da documentação.

## G.4.2. Outros modos do Emacs

O GNU Emacs vem com um modo SGML diferente, que não é tão poderoso quanto o PSGML, mas que é menos confuso e mais leve. Oferece, também, realce da sintaxe (bloqueio de fonte), que pode ser muito útil.

Norm Walsh disponibiliza um modo principal do GNU Emacs para editar documentos DocBook (<http://nwalsh.com/emacs/docbookide/index.html>), que também possui bloqueio de fonte e várias funcionalidades para reduzir a digitação.

## G.5. Guia de estilo

### G.5.1. Páginas de referência

As páginas de referência devem seguir uma organização padrão. Isto permite aos usuários encontrarem as informações desejadas mais rapidamente e, também, encoraja os autores a documentar todos os aspectos relevantes do comando. A consistência não é desejada apenas entre as páginas de referência do PostgreSQL, mas também com as páginas de referência disponibilizadas pelo sistema operacional e por outros pacotes. As diretrizes descritas a seguir foram desenvolvidas para esta finalidade. Em sua maior parte estas diretrizes são consistentes com as diretrizes semelhantes estabelecidas por vários sistemas operacionais.

As páginas de referência que descrevem comandos executáveis devem conter as seguintes seções, nesta ordem. As seções que não se aplicam podem ser omitidas. Seções adicionais de nível mais alto devem ser utilizadas apenas em circunstâncias especiais; geralmente esta informação pertence à seção “Utilização”.

Nome

Esta seção é gerada automaticamente. Contém o nome do comando e um breve resumo de sua funcionalidade.

Sinopse

Esta seção contém o diagrama da sintaxe do comando. Normalmente a sinopse não deve relacionar todas as opções de linha de comando; isto é feito abaixo. Em vez disto, devem ser relacionados os principais componentes da linha de comando, tal como o local dos arquivos de entrada e de saída.

Descrição

Vários parágrafos explicando o que o comando faz.

## Opções

Uma lista descrevendo cada opção de linha de comando. Havendo muitas opções, podem ser usadas subseções.

## Status de saída

Se o programa utilizar zero para execução bem-sucedida, e diferente de zero para mal-sucedida, então isto não precisa ser documentado. Havendo um significado por trás dos códigos de retorno diferentes de zero, estes devem ser descritos aqui.

## Utilização

Descreve todas as sub-linguagens ou interfaces em tempo de execução do programa. Normalmente esta seção pode ser omitida quando o programa não é interativo. Caso contrário, esta seção engloba a descrição de todas as funcionalidades em tempo de execução. Devem ser utilizadas subseções quando for apropriado.

## Ambiente

Relaciona todas as variáveis de ambiente que o programa pode utilizar. Deve ser completa; mesmo variáveis que parecem triviais, como `SHELL`, podem ser de interesse do usuário.

## Arquivos

Relaciona todos os arquivos que o programa pode acessar implicitamente, ou seja, não relaciona os arquivos de entrada e de saída especificados na linha de comando, mas relaciona os arquivos de configuração, etc.

## Diagnósticos

Explica qualquer saída não usual que o programa pode produzir. Deve-se evitar relacionar todas as mensagens de erro possíveis; isto dá muito trabalho e possui pouca utilidade prática. Mas se, por exemplo, as mensagens de erro possuem um formato padrão que o usuário pode analisar, este é o lugar para que isto seja explicado.

## Notas

Tudo que não é adequado em outro lugar, mas em particular erros, problemas na implementação, considerações sobre segurança, e questões de compatibilidade.

## Exemplos

Exemplos

## Histórico

Havendo marcos relevantes na história do programa, estes devem ser descritos aqui. Normalmente esta seção pode ser omitida.

## Consulte também

Referências cruzadas, relacionadas na seguinte ordem: outras páginas de referência de comandos do PostgreSQL, páginas de referência de comandos SQL do PostgreSQL, citação dos manuais do PostgreSQL, outras páginas de referência (por exemplo, sistema operacional, outros pacotes), outra documentação. Os itens do mesmo grupo devem estar em ordem alfabética.

As páginas de referência contendo comandos SQL devem possuir as seguintes seções: Nome, Sinopse, Descrição, Parâmetros, Saídas, Notas, Exemplos, Compatibilidade, Histórico e Consulte Também. A seção sobre Parâmetros é como a seção Opções, mas há maior liberdade sobre que cláusulas do comando podem ser relacionadas. A seção Saídas somente é necessária quando o comando retorna algo diferente da marca padrão de comando terminado. A seção Compatibilidade deve explicar a extensão da conformidade<sup>3</sup> do comando com o padrão SQL, ou informar com que outro sistema de banco de dados é compatível. A seção Consulte Também dos comandos SQL deve relacionar os comandos SQL antes da referência cruzada com os programas.

# Notas

1. `man pages` — Unix Manual Page - Uma parte da extensa documentação on-line do Unix. FOLDOC - Free On-Line Dictionary of Computing (<http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=man+pages>) (N. do T.)
2. Para gerar esta documentação foi necessário instalar também: `libc6-dev`, `openjade`, `jadetex` e `opensp`. (N. do T.)



3. conformidade — atributos do software que o tornam consonante com padrões ou convenções relacionadas à portabilidade. NBR 13596/1996, Tecnologia da Informação - Avaliação de produto de software - Características de qualidade e diretrizes para o seu uso, ABNT, Rio de Janeiro. (N. do T.)

## Apêndice H. Projetos externos

O PostgreSQL é um projeto de software complexo, e seu gerenciamento é difícil. Descobriu-se que muitas melhorias ao PostgreSQL poderiam ser feitas de forma mais eficiente quando desenvolvidas em separado do núcleo do projeto. Os projetos em separado podem ter times de desenvolvedores, listas de discussão, acompanhamento de erros, e programação de liberação próprios. Ao mesmo tempo que esta independência torna o desenvolvimento mais fácil, torna o trabalho do usuário mais difícil. Os usuários precisam procurar por melhorias do banco de dados que vêm ao encontro de suas necessidades. Esta seção descreve algumas das mais populares melhorias desenvolvidas externamente, e mostra como encontrá-las.

Muitos projetos relacionados com o PostgreSQL são hospedados no GBorg em <http://gborg.postgresql.org> ou no pgFoundry em <http://pgfoundry.org>. Existem outros projetos relacionados com o PostgreSQL hospedados em outros lugares, mas é necessária uma busca na Internet para localizá-los.<sup>1</sup>

### H.1. Interfaces desenvolvidas externamente

O PostgreSQL inclui muito poucas interfaces junto com a distribuição base. A libpq está incluída porque é a interface C primária, e muitas outras interfaces são construídas sobre esta. O ecpg é incluído porque está ligado à gramática do lado servidor e, portanto, é muito dependente da versão do banco de dados. Todas as outras interfaces são projetos independentes devendo ser instaladas em separado.

Algumas das interfaces mais populares são:

psqlODBC

Esta é a interface mais comum para os aplicativos Windows.

pgjdbc

A interface JDBC.

Npgsql

Interface .Net para os aplicativos Windows mais recentes.

libpqxx

A interface C++ nova.

libpq++

A interface C++ antiga.

pgperl

A interface Perl com uma API semelhante à libpq.

DBD-Pg

Uma interface Perl que utiliza a API DBD padrão.

pgtclng

A nova versão da interface Tcl.

pgtcl

A versão original da interface Tcl.

PyGreSQL

A biblioteca de interface com a linguagem Python.

Todas estas podem ser encontradas em GBorg (<http://gborg.postgresql.org>) ou em pgFoundry (<http://pgfoundry.org>).

## H.2. Extensões

Desde o princípio o PostgreSQL foi projetado para ser extensível. Por esta razão, as extensões carregadas no banco de dados podem funcionar exatamente como as funcionalidades incorporadas ao banco de dados. O diretório `contrib/` que acompanha o código fonte contém várias extensões. O arquivo `README` neste diretório contém um resumo. Estão incluídas ferramentas de conversão, indexação completa de texto, ferramentas XML, além de tipos de dado e métodos de indexação adicionais. Outras extensões são desenvolvidas de forma independente, como o PostGIS. Até mesmo as soluções de replicação do PostgreSQL são desenvolvidas externamente. Por exemplo, Slony-I é uma solução de replicação mestre/escravo desenvolvida independentemente do núcleo do projeto.

Existem várias ferramentas de administração disponíveis para o PostgreSQL. A mais popular é a pgAdmin, e existem várias outras ferramentas comerciais disponíveis.

## Notas

1. Existem projetos armazenados no SourceForge em <http://sourceforge.net> (<http://sourceforge.net/>), como este projeto de tradução para o português do Brasil (<http://sourceforge.net/projects/pgdocptbr/>), e o phpPgAdmin (<http://sourceforge.net/projects/phpPgAdmin/>) para administração do PostgreSQL via Web, além de outros. (N. do T.)