PostgreSQL 7.2.8 Documentation

The PostgreSQL Global Development Group
Copyright © 1996-2001 The PostgreSQL Global Development Group

Legal Notice

PostgreSQL is Copyright © 1996-2001 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.

Table of Contents

PostgreSQL 7.2.8 Tutorial x
Welcome xiii
1. Getting Started 1
1.1. Installation 1
1.2. Architectural Fundamentals 1
1.3. Creating a Database 2
1.4. Accessing a Database 3
2. The SQL Language 5
2.1. Introduction 5
2.2. Concepts 5
2.3. Creating a New Table 5
2.4. Populating a Table With Rows 6
2.5. Querying a Table 7
2.6. Joins Between Tables 8
2.7. Aggregate Functions 10
2.8. Updates 11
2.9. Deletions 12
3. Advanced Features 13
3.1. Introduction 13
3.2. Views 13
3.3. Foreign Keys 13
3.4. Transactions 14
3.5. Inheritance 15
3.6. Conclusion 17
PostgreSQL 7.2.8 User's Guide xviii
Preface xxvi
1. What is PostgreSQL? xxvi
2. A Short History of PostgreSQL xxvi
3. Documentation Resources xxviii
4. Terminology and Notation xxix
5. Bug Reporting Guidelines xxx
6. Y2K Statement xxxiii
1. SQL Syntax 1
1.1. Lexical Structure 1
1.2. Columns 6
1.3. Value Expressions 7
1.4. Lexical Precedence 10
2. Queries 12
2.1. Overview 12
2.2. Table Expressions 12
2.3. Select Lists 18
2.4. Combining Queries 19
2.5. Sorting Rows 20
2.6. LIMIT and OFFSET 20
3. Data Types 22
3.1. Numeric Types 23
3.2. Monetary Type 26
3.3. Character Types 27
3.4. Binary Strings 28
3.5. Date/Time Types 30
3.6. Boolean Type 36

3.7. Geometric Types 37
3.8. Network Address Data Types 40
3.9. Bit String Types 42
4. Functions and Operators 43
4.1. Logical Operators 43
4.2. Comparison Operators 43
4.3. Mathematical Functions and Operators 45
4.4. String Functions and Operators 47
4.5. Binary String Functions and Operators 50
4.6. Pattern Matching 51
4.7. Data Type Formatting Functions 55
4.8. Date/Time Functions and Operators 60
4.9. Geometric Functions and Operators 66
4.10. Network Address Type Functions 69
4.11. Sequence-Manipulation Functions 71
4.12. Conditional Expressions 72
4.13. Miscellaneous Functions 74
4.14. Aggregate Functions 76
4.15. Subquery Expressions 77
5. Type Conversion 82
5.1. Introduction 82
5.2. Overview 82
5.3. Operators 83
5.4. Functions 86
5.5. Query Targets 88
5.6. UNION and CASE Constructs 89
6. Arrays 91
7. Indexes 95
7.1. Introduction 95
7.2. Index Types 96
7.3. Multicolumn Indexes 96
7.4. Unique Indexes 97
7.5. Functional Indexes 97
7.6. Operator Classes 98
7.7. Keys 98
7.8. Partial Indexes 100
7.9. Examining Index Usage 102
8. Inheritance 104
9. Multiversion Concurrency Control 107
9.1. Introduction 107
9.2. Transaction Isolation 107
9.3. Read Committed Isolation Level 108
9.4. Serializable Isolation Level 108
9.5. Data consistency checks at the application level 109
9.6. Locking and Tables 109
9.7. Locking and Indexes 111
10. Managing a Database 112
10.1. Database Creation 112
10.2. Accessing a Database 112
10.3. Destroying a Database 113
11. Performance Tips 114
11.1. Using EXPLAIN 114
11.2. Statistics used by the Planner 117
11.3. Controlling the Planner with Explicit JOINs 120

11.4. Populating a Database 121
A. Date/Time Support 123
A.1. Date/Time Keywords 123
A.2. Time Zones 124
A.3. History of Units 128
B. SQL Key Words 130
Bibliography 146
PostgreSQL 7.2.8 Administrator's Guide cxlviii
1. Installation Instructions 1
1.1. Short Version 1
1.2. Requirements 1
1.3. Getting The Source 2
1.4. If You Are Upgrading 2
1.5. Installation Procedure 3
1.6. Post-Installation Setup 10
1.7. Supported Platforms 11
2. Installation on Windows 15
3. Server Runtime Environment 16
3.1. The PostgreSQL user account 16
3.2. Creating a database cluster 16
3.3. Starting the database server 17
3.4. Run-time configuration 20
3.5. Managing Kernel Resources 30
3.6. Shutting down the server 35
3.7. Secure TCP/IP Connections with SSL 36
3.8. Secure TCP/IP Connections with SSH tunnels 37
4. Client Authentication 38
4.1. The `pg_hba.conf` file 38
4.2. Authentication methods 42
4.3. Authentication problems 45
5. Localization 47
5.1. Locale Support 47
5.2. Multibyte Support 49
5.3. Single-byte character set recoding 56
6. Managing Databases 57
6.1. Creating a Database 57
6.2. Destroying a Database 59
7. Database Users and Permissions 61
7.1. Database Users 61
7.2. Groups 62
7.3. Privileges 62
7.4. Functions and Triggers 62
8. Routine Database Maintenance Tasks 64
8.1. General Discussion 64
8.2. Routine Vacuuming 64
8.3. Log File Maintenance 67
9. Backup and Restore 69
9.1. SQL Dump 69
9.2. File system level backup 71
9.3. Migration between releases 72
10. Monitoring Database Activity 73
10.1. Standard Unix Tools 73
10.2. Statistics Collector 74
11. Write-Ahead Logging (WAL) 78

11.1. General Description 78
11.2. Implementation 79
11.3. WAL Configuration 80
12. Disk Storage 82
13. Database Failures 83
13.1. Disk Filled 83
13.2. Disk Failed 83
14. Regression Tests 84
14.1. Introduction 84
14.2. Running the Tests 84
14.3. Test Evaluation 85
14.4. Platform-specific comparison files 87
A. Release Notes 89
A.1. Release 7.2.8 89
A.2. Release 7.2.7 89
A.3. Release 7.2.6 90
A.4. Release 7.2.5 91
A.5. Release 7.2.4 91
A.6. Release 7.2.3 92
A.7. Release 7.2.2 92
A.8. Release 7.2.1 93
A.9. Release 7.2 94
A.10. Release 7.1.3 105
A.11. Release 7.1.2 105
A.12. Release 7.1.1 106
A.13. Release 7.1 106
A.14. Release 7.0.3 111
A.15. Release 7.0.2 112
A.16. Release 7.0.1 112
A.17. Release 7.0 113
A.18. Release 6.5.3 121
A.19. Release 6.5.2 121
A.20. Release 6.5.1 122
A.21. Release 6.5 123
A.22. Release 6.4.2 128
A.23. Release 6.4.1 128
A.24. Release 6.4 129
A.25. Release 6.3.2 134
A.26. Release 6.3.1 135
A.27. Release 6.3 136
A.28. Release 6.2.1 141
A.29. Release 6.2 142
A.30. Release 6.1.1 144
A.31. Release 6.1 145
A.32. Release 6.0 148
A.33. Release 1.09 150
A.34. Release 1.02 150
A.35. Release 1.01 152
A.36. Release 1.0 155
A.37. Postgres95 Release 0.03 156
A.38. Postgres95 Release 0.02 158
A.39. Postgres95 Release 0.01 159
PostgreSQL 7.2.8 Programmer's Guide clx
I. Client Interfaces 1

1. libpq - C Library 5
2. Large Objects 35
3. libpq++ - C++ Binding Library 43
4. pgtcl - Tcl Binding Library 50
5. libpgeasy - Simplified C Library 71
6. ecpg - Embedded SQL in C 72
7. ODBC Interface 81
8. JDBC Interface 88
9. PyGreSQL - Python Interface 118
II. Server Programming 171
10. Architecture 174
11. Extending SQL: An Overview 177
12. Extending SQL: Functions 181
13. Extending SQL: Types 202
14. Extending SQL: Operators 204
15. Extending SQL: Aggregates 209
16. The Rule System 211
17. Interfacing Extensions To Indexes 234
18. Index Cost Estimation Functions 240
19. GiST Indexes 243
20. Triggers 245
21. Server Programming Interface 252
III. Procedural Languages 289
22. Procedural Languages 292
23. PL/pgSQL - SQL Procedural Language 294
24. PL/Tcl - Tcl Procedural Language 327
25. PL/Perl - Perl Procedural Language 334
26. PL/Python - Python Procedural Language 338
PostgreSQL 7.2.8 Reference Manual cccxli
Preface cccxlv
I. SQL Commands 346
ABORT 348
ALTER GROUP 349
ALTER TABLE 350
ALTER USER 354
ANALYZE 356
BEGIN 358
CHECKPOINT 360
CLOSE 361
CLUSTER 362
COMMENT 364
COMMIT 366
COPY 367
CREATE AGGREGATE 373
CREATE CONSTRAINT TRIGGER 376
CREATE DATABASE 377
CREATE FUNCTION 380
CREATE GROUP 384
CREATE INDEX 386
CREATE LANGUAGE 390
CREATE OPERATOR 393
CREATE RULE 397
CREATE SEQUENCE 400
CREATE TABLE 403

CREATE TABLE AS 411
CREATE TRIGGER 413
CREATE TYPE 416
CREATE USER 420
CREATE VIEW 423
DECLARE 425
DELETE 428
DROP AGGREGATE 430
DROP DATABASE 431
DROP FUNCTION 432
DROP GROUP 434
DROP INDEX 435
DROP LANGUAGE 436
DROP OPERATOR 437
DROP RULE 439
DROP SEQUENCE 440
DROP TABLE 441
DROP TRIGGER 443
DROP TYPE 445
DROP USER 447
DROP VIEW 448
END 450
EXPLAIN 451
FETCH 454
GRANT 457
INSERT 460
LISTEN 462
LOAD 464
LOCK 465
MOVE 470
NOTIFY 471
REINDEX 473
RESET 475
REVOKE 476
ROLLBACK 478
SELECT 479
SELECT INTO 490
SET 492
SET CONSTRAINTS 496
SET SESSION AUTHORIZATION 497
SET TRANSACTION 498
SHOW 500
TRUNCATE 501
UNLISTEN 502
UPDATE 504
VACUUM 506
II. PostgreSQL Client Applications 509
createdb 510
createlang 512
createuser 514
dropdb 517
droplang 519
dropuser 521
ecpg 523

pgaccess 527
pg_config 529
pg_dump 531
pg_dumpall 537
pg_restore 539
psql 545
pgtclsh 566
pgtksh 567
vacuumdb 568
III. PostgreSQL Server Applications 571
initdb 572
initlocation 574
ipcclean 575
pg_ctl 576
pg_passwd 579
postgres 580
postmaster 583
PostgreSQL 7.2.8 Developer's Guide ..... dlxxxvii
1. PostgreSQL Source Code 1
1.1. Formatting 1
2. Overview of PostgreSQL Internals 2
2.1. The Path of a Query 2
2.2. How Connections are Established 2
2.3. The Parser Stage 3
2.4. The PostgreSQL Rule System 5
2.5. Planner/Optimizer 6
2.6. Executor 7
3. System Catalogs 9
3.1. Overview 9
3.2. pg_aggregate 10
3.3. pg_attrdef 10
3.4. pg_attribute 11
3.5. pg_class 13
3.6. pg_database 15
3.7. pg_description 16
3.8. pg_group 17
3.9. pg_index 17
3.10. pg_inherits 18
3.11. pg_language 19
3.12. pg_largeobject 20
3.13. pg_listener 20
3.14. pg_operator 21
3.15. pg_proc 22
3.16. pg_relcheck 23
3.17. pg_rewrite 23
3.18. pg_shadow 24
3.19. pg_statistic 25
3.20. pg_trigger 26
3.21. pg_type 27
4. Frontend/Backend Protocol 32
4.1. Overview 32
4.2. Protocol 32
4.3. Message Data Types 38
4.4. Message Formats 38

5. gcc Default Optimizations 46
6. BKI Backend Interface 47
6.1. BKI File Format 47
6.2. BKI Commands 47
6.3. Example 48
7. Page Files 49
8. Genetic Query Optimization 50
8.1. Query Handling as a Complex Optimization Problem 50
8.2. Genetic Algorithms 50
8.3. Genetic Query Optimization (GEQO) in PostgreSQL 51
8.4. Further Readings 52
9. Native Language Support 53
9.1. For the Translator 53
9.2. For the Programmer 55
A. The CVS Repository 58
A.1. Getting The Source Via Anonymous CVS 58
A.2. CVS Tree Organization 59
A.3. Getting The Source Via CVSup 60
B. Documentation 66
B.1. DocBook 66
B.2. Toolsets 66
B.3. Building The Documentation 70
B.4. Documentation Authoring 74

PostgreSQL 7.2.8 Tutorial

The PostgreSQL Global Development Group

PostgreSQL 7.2.8 Tutorial

The PostgreSQL Global Development Group
Copyright © 1996-2001 The PostgreSQL Global Development Group

Legal Notice

PostgreSQL is Copyright © 1996-2001 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.

Table of Contents

Welcome xiii
1. Getting Started 1
    1.1. Installation 1
    1.2. Architectural Fundamentals 1
    1.3. Creating a Database 2
    1.4. Accessing a Database 3
2. The SQL Language 5
    2.1. Introduction 5
    2.2. Concepts 5
    2.3. Creating a New Table 5
    2.4. Populating a Table With Rows 6
    2.5. Querying a Table 7
    2.6. Joins Between Tables 8
    2.7. Aggregate Functions 10
    2.8. Updates 11
    2.9. Deletions 12
3. Advanced Features 13
    3.1. Introduction 13
    3.2. Views 13
    3.3. Foreign Keys 13
    3.4. Transactions 14
    3.5. Inheritance 15
    3.6. Conclusion 17

Welcome

Welcome to PostgreSQL and the *PostgreSQL Tutorial*. The following few chapters are intended to give a simple introduction to PostgreSQL, relational database concepts, and the SQL language to those who are new to any one of these aspects. We only assume some general knowledge about how to use computers. No particular Unix or programming experience is required.

After you have worked through this tutorial you might want to move on to reading the PostgreSQL 7.2.8 User's Guide to gain a more formal knowledge of the SQL language, or the PostgreSQL 7.2.8 Programmer's Guide for information about developing applications for PostgreSQL.

We hope you have a pleasant experience with PostgreSQL.

Chapter 1. Getting Started

1.1. Installation

Before you can use PostgreSQL you need to install it, of course. It is possible that PostgreSQL is already installed at your site, either because it was included in your operating system distribution or because the system administrator already installed it. If that is the case, you should obtain information from the operating system documentation or your system administrator about how to access PostgreSQL.

If you are not sure whether PostgreSQL is already available or whether you can use it for your experimentation then you can install it yourself. Doing so is not hard and it can be a good exercise. PostgreSQL can be installed by any unprivileged user, no superuser (root) access is required.

If you are installing PostgreSQL yourself, then refer to the *Administrator's Guide* for instructions on installation, and return to this guide when the installation is complete. Be sure to follow closely the section about setting up the appropriate environment variables.

If your site administrator has not set things up in the default way, you may have some more work to do. For example, if the database server machine is a remote machine, you will need to set the `PGHOST` environment variable to the name of the database server machine. The environment variable `PGPORT` may also have to be set. The bottom line is this: if you try to start an application program and it complains that it cannot connect to the database, you should consult your site administrator or, if that is you, the documentation to make sure that your environment is properly set up. If you did not understand the preceding paragraph then read the next section.

1.2. Architectural Fundamentals

Before we proceed, you should understand the basic PostgreSQL system architecture. Understanding how the parts of PostgreSQL interact will make this chapter somewhat clearer.

In database jargon, PostgreSQL uses a client/server model. A PostgreSQL session consists of the following cooperating processes (programs):

- A server process, which manages the database files, accepts connections to the database from client applications, and performs actions on the database on behalf of the clients. The database server program is called `postmaster`.
- The user's client (frontend) application that wants to perform database operations. Client applications can be very diverse in nature: They could be a text-oriented tool, a graphical application, a web server that accesses the database to display web pages, or a specialized database maintenance tool. Some client applications are supplied with the PostgreSQL distribution, most are developed by users.

As is typical of client/server applications, the client and the server can be on different hosts. In that case they communicate over a TCP/IP network connection. You should keep this in mind, because the files that can be accessed on a client machine might not be accessible (or might only be accessible using a different file name) on the database server machine.

The PostgreSQL server can handle multiple concurrent connections from clients. For that purpose it starts (“forks”) a new process for each connection. From that point on, the client and the new server process communicate without intervention by the original `postmaster` process. Thus, the `postmaster` is always running, waiting for client connections, whereas client and associated server processes come and go. (All of this is of course invisible to the user. We only mention it here for completeness.)

1.3. Creating a Database

The first test to see whether you can access the database server is to try to create a database. A running PostgreSQL server can manage many databases. Typically, a separate database is used for each project or for each user.

Possibly, your site administrator has already created a database for your use. He should have told you what the name of your database is. In this case you can omit this step and skip ahead to the next section.

To create a new database, in this example named `mydb`, you use the following command:

```
$ createdb mydb
```

This should produce as response:

```
CREATE DATABASE
```

If so, this step was successful and you can skip over the remainder of this section.

If you see a message similar to

```
createdb: command not found
```

then PostgreSQL was not installed properly. Either it was not installed at all or the search path was not set correctly. Try calling the command with an absolute path instead:

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

The path at your site might be different. Contact your site administrator or check back in the installation instructions to correct the situation.

Another response could be this:

```
psql: could not connect to server: Connection refused
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
createdb: database creation failed
```

This means that the server was not started, or it was not started where `createdb` expected it. Again, check the installation instructions or consult the administrator.

If you do not have the privileges required to create a database, you will see the following:

```
ERROR:  CREATE DATABASE: permission denied
createdb: database creation failed
```

Not every user has authorization to create new databases. If PostgreSQL refuses to create databases for you then the site administrator needs to grant you permission to create databases. Consult your site administrator if this occurs. If you installed PostgreSQL yourself then you should log in for the purposes of this tutorial under the user account that you started the server as. [1]

[1] As an explanation for why this works: PostgreSQL user names are separate from operating system user accounts. If you connect to a database, you can choose what PostgreSQL user name to connect as; if you don't, it will default to the same name as your current operating system account. As it happens, there will always be a PostgreSQL user account that has the same name as the operating system user that started the server, and it also happens that that user always has permission to create databases. Instead of logging in as that user you can also specify the `-U` option everywhere to select a PostgreSQL user name to connect as.

You can also create databases with other names. PostgreSQL allows you to create any number of databases at a given site. Database names must have an alphabetic first character and are limited to 31 characters in length. A convenient choice is to create a database with the same name as your current user name. Many tools assume that database name as the default, so it can save you some typing. To create that database, simply type

```
$ createdb
```

If you don't want to use your database anymore you can remove it. For example, if you are the owner (creator) of the database `mydb`, you can destroy it using the following command:

```
$ dropdb mydb
```

(For this command, the database name does not default to the user account name. You always need to specify it.) This action physically removes all files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

1.4. Accessing a Database

Once you have created a database, you can access it by:

- Running the PostgreSQL interactive terminal program, called *psql*, which allows you to interactively enter, edit, and execute SQL commands.
- Using an existing graphical frontend tool like PgAccess or ApplixWare (via ODBC) to create and manipulate a database. These possibilities are not covered in this tutorial.
- Writing a custom application, using one of the several available language bindings. These possibilities are discussed further in *The PostgreSQL Programmer's Guide*.

You probably want to start up `psql`, to try out the examples in this tutorial. It can be activated for the `mydb` database by typing the command:

```
$ psql mydb
```

If you leave off the database name then it will default to your user account name. You already discovered this scheme in the previous section.

In `psql`, you will be greeted with the following message:

```
Welcome to psql, 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

mydb=>
```

The last line could also be

```
mydb=#
```

That would mean you are a database superuser, which is most likely the case if you installed PostgreSQL yourself. Being a superuser means that you are not subject to access controls. For the purpose of this tutorial this is not of importance.

If you have encountered problems starting psql then go back to the previous section. The diagnostics of psql and createdb are similar, and if the latter worked the former should work as well.

The last line printed out by psql is the prompt, and it indicates that psql is listening to you and that you can type SQL queries into a work space maintained by psql. Try out these commands:

```
mydb=> SELECT version();
                                version
---------------------------------------------------------------------
 PostgreSQL 7.2devel on i586-pc-linux-gnu, compiled by GCC 2.96
(1 row)

mydb=> SELECT current_date;
    date
------------
 2001-08-31
(1 row)

mydb=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

The psql program has a number of internal commands that are not SQL commands. They begin with the backslash character, "\". Some of these commands were listed in the welcome message. For example, you can get help on the syntax of various PostgreSQL SQL commands by typing:

```
mydb=> \h
```

To get out of psql, type

```
mydb=> \q
```

and psql will quit and return you to your command shell. (For more internal commands, type \? at the psql prompt.) The full capabilities of psql are documented in the *Reference Manual*. If PostgreSQL is installed correctly you can also type man psql at the operating system shell prompt to see the documentation. In this tutorial we will not use these features explicitly, but you can use them yourself when you see fit.

Chapter 2. The SQL Language

2.1. Introduction

This chapter provides an overview of how to use SQL to perform simple operations. This tutorial is only intended to give you an introduction and is in no way a complete tutorial on SQL. Numerous books have been written on SQL92, including [melt93] and [date97]. You should be aware that some PostgreSQL language features are extensions to the standard.

In the examples that follow, we assume that you have created a database named `mydb`, as described in the previous chapter, and have started psql.

Examples in this manual can also be found in the PostgreSQL source distribution in the directory `src/tutorial/`. Refer to the `README` file in that directory for how to use them. To start the tutorial, do the following:

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

mydb=> \i basics.sql
```

The `\i` command reads in commands from the specified file. The `-s` option puts you in single step mode which pauses before sending each query to the server. The commands used in this section are in the file `basics.sql`.

2.2. Concepts

PostgreSQL is a *relational database management system* (RDBMS). That means it is a system for managing data stored in *relations*. Relation is essentially a mathematical term for *table*. The notion of storing data in tables is so commonplace today that it might seem inherently obvious, but there are a number of other ways of organizing databases. Files and directories on Unix-like operating systems form an example of a hierarchical database. A more modern development is the object-oriented database.

Each table is a named collection of *rows*. Each row of a given table has the same set of named *columns*, and each column is of a specific data type. Whereas columns have a fixed order in each row, it is important to remember that SQL does not guarantee the order of the rows within the table in any way (although they can be explicitly sorted for display).

Tables are grouped into databases, and a collection of databases managed by a single PostgreSQL server instance constitutes a database *cluster*.

2.3. Creating a New Table

You can create a new table by specifying the table name, along with all column names and their types:

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- low temperature
    temp_hi         int,           -- high temperature
    prcp            real,          -- precipitation
```

```
    date            date
);
```

You can enter this into `psql` with the line breaks. `psql` will recognize that the command is not terminated until the semicolon.

White space (i.e., spaces, tabs, and newlines) may be used freely in SQL commands. That means you can type the command aligned differently than above, or even all on one line. Two dashes ("`--`") introduce comments. Whatever follows them is ignored up to the end of the line. SQL is case insensitive about key words and identifiers, except when identifiers are double-quoted to preserve the case (not done above).

`varchar(80)` specifies a data type that can store arbitrary character strings up to 80 characters in length. `int` is the normal integer type. `real` is a type for storing single precision floating-point numbers. `date` should be self-explanatory. (Yes, the column of type `date` is also named `date`. This may be convenient or confusing -- you choose.)

PostgreSQL supports the usual SQL types `int`, `smallint`, `real`, `double precision`, `char(`*N*`)`, `varchar(`*N*`)`, `date`, `time`, `timestamp`, and `interval`, as well as other types of general utility and a rich set of geometric types. PostgreSQL can be customized with an arbitrary number of user-defined data types. Consequently, type names are not syntactical keywords, except where required to support special cases in the SQL standard.

The second example will store cities and their associated geographical location:

```
CREATE TABLE cities (
    name            varchar(80),
    location        point
);
```

The `point` type is an example of a PostgreSQL-specific data type.

Finally, it should be mentioned that if you don't need a table any longer or want to recreate it differently you can remove it using the following command:

```
DROP TABLE tablename;
```

2.4. Populating a Table With Rows

The `INSERT` statement is used to populate a table with rows:

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

Note that all data types use rather obvious input formats. Constants that are not simple numeric values usually must be surrounded by single quotes (`'`), as in the example. The `date` column is actually quite flexible in what it accepts, but for this tutorial we will stick to the unambiguous format shown here.

The `point` type requires a coordinate pair as input, as shown here:

```
INSERT INTO cities  VALUES ('San Francisco', '(-194.0, 53.0)');
```

The syntax used so far requires you to remember the order of the columns. An alternative syntax allows you to list the columns explicitly:

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
    VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

You can list the columns in a different order if you wish or even omit some columns, e.g., if the precipitation is unknown:

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Many developers consider explicitly listing the columns better style than relying on the order implicitly.

Please enter all the commands shown above so you have some data to work with in the following sections.

You could also have used `COPY` to load large amounts of data from flat-text files. This is usually faster because the `COPY` command is optimized for this application while allowing less flexibility than `INSERT`. An example would be:

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

where the file name for the source file must be available to the backend server machine, not the client, since the backend server reads the file directly. You can read more about the `COPY` command in the *Reference Manual*.

2.5. Querying a Table

To retrieve data from a table, the table is *queried*. An SQL `SELECT` statement is used to do this. The statement is divided into a select list (the part that lists the columns to be returned), a table list (the part that lists the tables from which to retrieve the data), and an optional qualification (the part that specifies any restrictions). For example, to retrieve all the rows of table `weather`, type:

```
SELECT * FROM weather;
```

(here `*` means "all columns") and the output should be:

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      43 |      57 |    0 | 1994-11-29
 Hayward       |      37 |      54 |      | 1994-11-29
(3 rows)
```

You may specify any arbitrary expressions in the target list. For example, you can do:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

This should give:

```
     city      | temp_avg |    date
---------------+----------+------------
 San Francisco |       48 | 1994-11-27
 San Francisco |       50 | 1994-11-29
 Hayward       |       45 | 1994-11-29
(3 rows)
```

Notice how the `AS` clause is used to relabel the output column. (It is optional.)

Arbitrary Boolean operators (`AND`, `OR`, and `NOT`) are allowed in the qualification of a query. For example, the following retrieves the weather of San Francisco on rainy days:

```
SELECT * FROM weather
    WHERE city = 'San Francisco'
    AND prcp > 0.0;
```

Result:

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
(1 row)
```

As a final note, you can request that the results of a select can be returned in sorted order or with duplicate rows removed. (Just to make sure the following won't confuse you, `DISTINCT` and `ORDER BY` can be used separately.)

```
SELECT DISTINCT city
    FROM weather
    ORDER BY city;

     city
---------------
 Hayward
 San Francisco
(2 rows)
```

2.6. Joins Between Tables

Thus far, our queries have only accessed one table at a time. Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. A query that accesses multiple rows of the same or different tables at one time is called a *join* query. As an example, say you wish to list all the weather records together with the location of the associated city. To do that, we need to compare the city column of each row of the weather table with the name column of all rows in the cities table, and select the pairs of rows where these values match.

Note

This is only a conceptual model. The actual join may be performed in a more efficient manner, but this is invisible to the user.

This would be accomplished by the following query:

```
SELECT *
    FROM weather, cities
    WHERE city = name;

     city      | temp_lo | temp_hi | prcp |    date    |     name
 | location
---------------+---------+---------+------+------------
+---------------+-----------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco
 | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco
 | (-194,53)
(2 rows)
```

Observe two things about the result set:

- There is no result row for the city of Hayward. This is because there is no matching entry in the `cities` table for Hayward, so the join ignores the unmatched rows in the weather table. We will see shortly how this can be fixed.
- There are two columns containing the city name. This is correct because the lists of columns of the `weather` and the `cities` table are concatenated. In practice this is undesirable, though, so you will probably want to list the output columns explicitly rather than using `*`:

```
SELECT city, temp_lo, temp_hi, prcp, date, location
    FROM weather, cities
    WHERE city = name;
```

Exercise: Attempt to find out the semantics of this query when the `WHERE` clause is omitted.

Since the columns all had different names, the parser automatically found out which table they belong to, but it is good style to fully qualify column names in join queries:

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
    FROM weather, cities
    WHERE cities.name = weather.city;
```

Join queries of the kind seen thus far can also be written in this alternative form:

```
SELECT *
    FROM weather INNER JOIN cities ON (weather.city = cities.name);
```

This syntax is not as commonly used as the one above, but we show it here to help you understand the following topics.

Now we will figure out how we can get the Hayward records back in. What we want the query to do is to scan the `weather` table and for each row to find the matching `cities` row. If no matching row is found we want some “empty values” to be substituted for the `cities` table's columns. This kind of query is called an *outer join*. (The joins we have seen so far are inner joins.) The command looks like this:

```
SELECT *
    FROM weather LEFT OUTER JOIN cities ON (weather.city =
 cities.name);

     city      | temp_lo | temp_hi | prcp |    date    |     name
 | location
---------------+---------+---------+------+------------
+---------------+-----------
 Hayward       |      37 |      54 |      | 1994-11-29 |
  |
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco
 | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco
 | (-194,53)
(3 rows)
```

This query is called a *left outer join* because the table mentioned on the left of the join operator will have each of its rows in the output at least once, whereas the table on the right will only have those rows output that match some row of the left table. When outputting a left-table row for which there is no right-table match, empty (NULL) values are substituted for the right-table columns.

Exercise: There are also right outer joins and full outer joins. Try to find out what those do.

We can also join a table against itself. This is called a *self join*. As an example, suppose we wish to find all the weather records that are in the temperature range of other weather records. So we need to compare the `temp_lo` and `temp_hi` columns of each `weather` row to the `temp_lo` and `temp_hi` columns of all other `weather` rows. We can do this with the following query:

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
    W2.city, W2.temp_lo AS low, W2.temp_hi AS high
    FROM weather W1, weather W2
    WHERE W1.temp_lo < W2.temp_lo
    AND W1.temp_hi > W2.temp_hi;

     city      | low | high |     city      | low | high
---------------+-----+------+---------------+-----+------
 San Francisco |  43 |   57 | San Francisco |  46 |   50
 Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 rows)
```

Here we have relabeled the weather table as `W1` and `W2` to be able to distinguish the left and right side of the join. You can also use these kinds of aliases in other queries to save some typing, e.g.:

```
SELECT *
    FROM weather w, cities c
    WHERE w.city = c.name;
```

You will encounter this style of abbreviating quite frequently.

2.7. Aggregate Functions

Like most other relational database products, PostgreSQL supports aggregate functions. An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the `count`, `sum`, `avg` (average), `max` (maximum) and `min` (minimum) over a set of rows.

As an example, we can find the highest low-temperature reading anywhere with

```
SELECT max(temp_lo) FROM weather;

 max
-----
  46
(1 row)
```

If we want to know what city (or cities) that reading occurred in, we might try

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);     WRONG
```

but this will not work since the aggregate `max` cannot be used in the `WHERE` clause. (This restriction exists because the `WHERE` clause determines the rows that will go into the aggregation stage; so it has to be evaluated before aggregate functions are computed.) However, as is often the case the query can be restated to accomplish the intended result; here by using a *subquery*:

```
SELECT city FROM weather
    WHERE temp_lo = (SELECT max(temp_lo) FROM weather);

     city
---------------
```

```
 San Francisco
(1 row)
```

This is OK because the sub-select is an independent computation that computes its own aggregate separately from what is happening in the outer select.

Aggregates are also very useful in combination with `GROUP BY` clauses. For example, we can get the maximum low temperature observed in each city with

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city;

     city      | max
---------------+-----
 Hayward       |  37
 San Francisco |  46
(2 rows)
```

which gives us one output row per city. Each aggregate result is computed over the table rows matching that city. We can filter these grouped rows using `HAVING`:

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city
    HAVING max(temp_lo) < 40;

  city   | max
---------+-----
 Hayward |  37
(1 row)
```

which gives us the same results for only the cities that have all `temp_lo` values below 40. Finally, if we only care about cities whose names begin with "`S`", we might do

```
SELECT city, max(temp_lo)
    FROM weather
    WHERE city LIKE 'S%'
    GROUP BY city
    HAVING max(temp_lo) < 40;
```

It is important to understand the interaction between aggregates and SQL's `WHERE` and `HAVING` clauses. The fundamental difference between `WHERE` and `HAVING` is this: `WHERE` selects input rows before groups and aggregates are computed (thus, it controls which rows go into the aggregate computation), whereas `HAVING` selects group rows after groups and aggregates are computed. Thus, the `WHERE` clause must not contain aggregate functions; it makes no sense to try to use an aggregate to determine which rows will be inputs to the aggregates. On the other hand, `HAVING` clauses always contain aggregate functions. (Strictly speaking, you are allowed to write a `HAVING` clause that doesn't use aggregates, but it's wasteful: The same condition could be used more efficiently at the `WHERE` stage.)

Observe that we can apply the city name restriction in `WHERE`, since it needs no aggregate. This is more efficient than adding the restriction to `HAVING`, because we avoid doing the grouping and aggregate calculations for all rows that fail the `WHERE` check.

2.8. Updates

You can update existing rows using the `UPDATE` command. Suppose you discover the temperature readings are all off by 2 degrees as of November 28, you may update the data as follows:

```
UPDATE weather
    SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
    WHERE date > '1994-11-28';
```

Look at the new state of the data:

```
SELECT * FROM weather;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
 Hayward       |      35 |      52 |      | 1994-11-29
(3 rows)
```

2.9. Deletions

Suppose you are no longer interested in the weather of Hayward, then you can do the following to delete those rows from the table. Deletions are performed using the `DELETE` command:

```
DELETE FROM weather WHERE city = 'Hayward';
```

All weather records belonging to Hayward are removed.

```
SELECT * FROM weather;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
(2 rows)
```

One should be wary of queries of the form

```
DELETE FROM tablename;
```

Without a qualification, `DELETE` will remove *all* rows from the given table, leaving it empty. The system will not request confirmation before doing this!

Chapter 3. Advanced Features

3.1. Introduction

In the previous chapter we have covered the basics of using SQL to store and access your data in PostgreSQL. We will now discuss some more advanced features of SQL that simplify management and prevent loss or corruption of your data. Finally, we will look at some PostgreSQL extensions.

This chapter will on occasion refer to examples found in Chapter 2, *The SQL Language* to change or improve them, so it will be of advantage if you have read that chapter. Some examples from this chapter can also be found in `advanced.sql` in the tutorial directory. This file also contains some example data to load, which is not repeated here. (Refer to Section 2.1, “Introduction” for how to use the file.)

3.2. Views

Refer back to the queries in Section 2.6, “Joins Between Tables”. Suppose the combined listing of weather records and city location is of particular interest to your application, but you don't want to type the query each time you need it. You can create a *view* over the query, which gives a name to the query that you can refer to like an ordinary table.

```
CREATE VIEW myview AS
    SELECT city, temp_lo, temp_hi, prcp, date, location
        FROM weather, cities
        WHERE city = name;

SELECT * FROM myview;
```

Making liberal use of views is a key aspect of good SQL database design. Views allow you to encapsulate the details of the structure of your tables, which may change as your application evolves, behind consistent interfaces.

Views can be used in almost any place a real table can be used. Building views upon other views is not uncommon.

3.3. Foreign Keys

Recall the `weather` and `cities` tables from Chapter 2, *The SQL Language*. Consider the following problem: You want to make sure that no one can insert rows in the `weather` table that do not have a matching entry in the `cities` table. This is called maintaining the *referential integrity* of your data. In simplistic database systems this would be implemented (if at all) by first looking at the `cities` table to check if a matching record exists, and then inserting or rejecting the new `weather` records. This approach has a number of problems and is very inconvenient, so PostgreSQL can do this for you.

The new declaration of the tables would look like this:

```
CREATE TABLE cities (
 city  varchar(80) primary key,
 location point
);

CREATE TABLE weather (
```

```
 city  varchar(80) references cities,
 temp_lo  int,
 temp_hi  int,
 prcp  real,
 date  date
);
```

Now try inserting an invalid record:

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

ERROR:  <unnamed> referential integrity violation - key referenced
 from weather not found in cities
```

The behavior of foreign keys can be finely tuned to your application. We will not go beyond this simple example in this tutorial, but just refer you to the *Reference Manual* for more information. Making correct use of foreign keys will definitely improve the quality of your database applications, so you are strongly encouraged to learn about them.

3.4. Transactions

Transactions are a fundamental concept of all database systems. The essential point of a transaction is that it bundles multiple steps into a single, all-or-nothing operation. The intermediate states between the steps are not visible to other concurrent transactions, and if some failure occurs that prevents the transaction from completing, then none of the steps affect the database at all.

For example, consider a bank database that contains balances for various customer accounts, as well as total deposit balances for branches. Suppose that we want to record a payment of $100.00 from Alice's account to Bob's account. Simplifying outrageously, the SQL commands for this might look like

```
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name =
 'Alice');
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name =
 'Bob');
```

The details of these commands are not important here; the important point is that there are several separate updates involved to accomplish this rather simple operation. Our bank's officers will want to be assured that either all these updates happen, or none of them happen. It would certainly not do for a system failure to result in Bob receiving $100.00 that was not debited from Alice. Nor would Alice long remain a happy customer if she was debited without Bob being credited. We need a guarantee that if something goes wrong partway through the operation, none of the steps executed so far will take effect. Grouping the updates into a *transaction* gives us this guarantee. A transaction is said to be *atomic*: from the point of view of other transactions, it either happens completely or not at all.

We also want a guarantee that once a transaction is completed and acknowledged by the database system, it has indeed been permanently recorded and won't be lost even if a crash ensues shortly thereafter. For example, if we are recording a cash withdrawal by Bob, we do not want any chance that the debit to his account will disappear in a crash just as he walks out the bank door. A transactional database guarantees that

all the updates made by a transaction are logged in permanent storage (i.e., on disk) before the transaction is reported complete.

Another important property of transactional databases is closely related to the notion of atomic updates: when multiple transactions are running concurrently, each one should not be able to see the incomplete changes made by others. For example, if one transaction is busy totalling all the branch balances, it would not do for it to include the debit from Alice's branch but not the credit to Bob's branch, nor vice versa. So transactions must be all-or-nothing not only in terms of their permanent effect on the database, but also in terms of their visibility as they happen. The updates made so far by an open transaction are invisible to other transactions until the transaction completes, whereupon all the updates become visible simultaneously.

In PostgreSQL, a transaction is set up by surrounding the SQL commands of the transaction with `BEGIN` and `COMMIT` commands. So our banking transaction would actually look like

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
-- etc etc
COMMIT;
```

If, partway through the transaction, we decide we don't want to commit (perhaps we just noticed that Alice's balance went negative), we can issue the command `ROLLBACK` instead of `COMMIT`, and all our updates so far will be canceled.

PostgreSQL actually treats every SQL statement as being executed within a transaction. If you don't issue a `BEGIN` command, then each individual statement has an implicit `BEGIN` and (if successful) `COMMIT` wrapped around it. A group of statements surrounded by `BEGIN` and `COMMIT` is sometimes called a *transaction block*.

Note

Some client libraries issue `BEGIN` and `COMMIT` commands automatically, so that you may get the effect of transaction blocks without asking. Check the documentation for the interface you are using.

3.5. Inheritance

Inheritance is a concept from object-oriented databases. It opens up interesting new possibilities of database design.

Let's create two tables: A table `cities` and a table `capitals`. Naturally, capitals are also cities, so you want some way to show the capitals implicitly when you list all cities. If you're really clever you might invent some scheme like this:

```
CREATE TABLE capitals (
    name            text,
    population      real,
    altitude        int,    -- (in ft)
    state           char(2)
);

CREATE TABLE non_capitals (
    name            text,
    population      real,
```

```
    altitude        int     -- (in ft)
);

CREATE VIEW cities AS
    SELECT name, population, altitude FROM capitals
        UNION
    SELECT name, population, altitude FROM non_capitals;
```

This works OK as far as querying goes, but it gets ugly when you need to update several rows, to name one thing.

A better solution is this:

```
CREATE TABLE cities (
    name            text,
    population      real,
    altitude        int     -- (in ft)
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

In this case, a row of `capitals` *inherits* all columns (`name`, `population`, and `altitude`) from its *parent*, `cities`. The type of the column `name` is `text`, a native PostgreSQL type for variable length character strings. State capitals have an extra column, state, that shows their state. In PostgreSQL, a table can inherit from zero or more other tables.

For example, the following query finds the names of all cities, including state capitals, that are located at an altitude over 500 ft.:

```
SELECT name, altitude
    FROM cities
    WHERE altitude > 500;
```

which returns:

```
   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
(3 rows)
```

On the other hand, the following query finds all the cities that are not state capitals and are situated at an altitude of 500 ft. or higher:

```
SELECT name, altitude
    FROM ONLY cities
    WHERE altitude > 500;

   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
(2 rows)
```

Here the `ONLY` before `cities` indicates that the query should be run over only the `cities` table, and not tables below `cities` in the inheritance hierarchy. Many of the commands that we have already discussed -- `SELECT`, `UPDATE` and `DELETE` -- support this `ONLY` notation.

3.6. Conclusion

PostgreSQL has many features not touched upon in this tutorial introduction, which has been oriented toward newer users of SQL. These features are discussed in more detail in both the *User's Guide* and the *Programmer's Guide*.

If you feel you need more introductory material, please visit the PostgreSQL web site[1] for links to more resources.

[1] http://www.postgresql.org

PostgreSQL 7.2.8 User's Guide

The PostgreSQL Global Development Group

PostgreSQL 7.2.8 User's Guide

The PostgreSQL Global Development Group
Copyright © 1996-2001 The PostgreSQL Global Development Group

Legal Notice

PostgreSQL is Copyright © 1996-2001 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.

Table of Contents

Preface xxvi
1. What is PostgreSQL? xxvi
2. A Short History of PostgreSQL xxvi
2.1. The Berkeley POSTGRES Project xxvii
2.2. Postgres95 xxvii
2.3. PostgreSQL xxviii
3. Documentation Resources xxviii
4. Terminology and Notation xxix
5. Bug Reporting Guidelines xxx
5.1. Identifying Bugs xxx
5.2. What to report xxxi
5.3. Where to report bugs xxxii
6. Y2K Statement xxxiii
1. SQL Syntax 1
1.1. Lexical Structure 1
1.1.1. Identifiers and Key Words 1
1.1.2. Constants 2
1.1.3. Operators 4
1.1.4. Special Characters 5
1.1.5. Comments 5
1.2. Columns 6
1.3. Value Expressions 7
1.3.1. Column References 7
1.3.2. Positional Parameters 8
1.3.3. Operator Invocations 8
1.3.4. Function Calls 8
1.3.5. Aggregate Expressions 8
1.3.6. Type Casts 9
1.3.7. Scalar Subqueries 9
1.4. Lexical Precedence 10
2. Queries 12
2.1. Overview 12
2.2. Table Expressions 12
2.2.1. FROM clause 12
2.2.2. WHERE clause 16
2.2.3. GROUP BY and HAVING clauses 17
2.3. Select Lists 18
2.3.1. Column Labels 18
2.3.2. DISTINCT 19
2.4. Combining Queries 19
2.5. Sorting Rows 20
2.6. LIMIT and OFFSET 20
3. Data Types 22
3.1. Numeric Types 23
3.1.1. The Integer Types 24
3.1.2. Arbitrary Precision Numbers 24
3.1.3. Floating-Point Types 25
3.1.4. The Serial Types 26
3.2. Monetary Type 26
3.3. Character Types 27
3.4. Binary Strings 28

3.5. Date/Time Types 30
3.5.1. Date/Time Input 31
3.5.2. Date/Time Output 35
3.5.3. Time Zones 35
3.5.4. Internals 36
3.6. Boolean Type 36
3.7. Geometric Types 37
3.7.1. Point 38
3.7.2. Line Segment 38
3.7.3. Box 38
3.7.4. Path 39
3.7.5. Polygon 39
3.7.6. Circle 40
3.8. Network Address Data Types 40
3.8.1. inet 40
3.8.2. cidr 41
3.8.3. inet vs cidr 41
3.8.4. macaddr 41
3.9. Bit String Types 42
4. Functions and Operators 43
4.1. Logical Operators 43
4.2. Comparison Operators 43
4.3. Mathematical Functions and Operators 45
4.4. String Functions and Operators 47
4.5. Binary String Functions and Operators 50
4.6. Pattern Matching 51
4.6.1. Pattern Matching with LIKE 52
4.6.2. POSIX Regular Expressions 52
4.7. Data Type Formatting Functions 55
4.8. Date/Time Functions and Operators 60
4.8.1. EXTRACT, date_part 61
4.8.2. date_trunc 64
4.8.3. Current Date/Time 65
4.9. Geometric Functions and Operators 66
4.10. Network Address Type Functions 69
4.11. Sequence-Manipulation Functions 71
4.12. Conditional Expressions 72
4.13. Miscellaneous Functions 74
4.14. Aggregate Functions 76
4.15. Subquery Expressions 77
5. Type Conversion 82
5.1. Introduction 82
5.2. Overview 82
5.3. Operators 83
5.4. Functions 86
5.5. Query Targets 88
5.6. UNION and CASE Constructs 89
6. Arrays 91
7. Indexes 95
7.1. Introduction 95
7.2. Index Types 96
7.3. Multicolumn Indexes 96
7.4. Unique Indexes 97
7.5. Functional Indexes 97

7.6. Operator Classes 98
7.7. Keys 98
7.8. Partial Indexes 100
7.9. Examining Index Usage 102
8. Inheritance 104
9. Multiversion Concurrency Control 107
9.1. Introduction 107
9.2. Transaction Isolation 107
9.3. Read Committed Isolation Level 108
9.4. Serializable Isolation Level 108
9.5. Data consistency checks at the application level 109
9.6. Locking and Tables 109
9.6.1. Table-level locks 109
9.6.2. Row-level locks 110
9.7. Locking and Indexes 111
10. Managing a Database 112
10.1. Database Creation 112
10.2. Accessing a Database 112
10.3. Destroying a Database 113
11. Performance Tips 114
11.1. Using EXPLAIN 114
11.2. Statistics used by the Planner 117
11.3. Controlling the Planner with Explicit JOINs 120
11.4. Populating a Database 121
11.4.1. Disable Autocommit 121
11.4.2. Use COPY FROM 121
11.4.3. Remove Indexes 122
11.4.4. ANALYZE Afterwards 122
A. Date/Time Support 123
A.1. Date/Time Keywords 123
A.2. Time Zones 124
A.2.1. Australian Time Zones 127
A.2.2. Date/Time Input Interpretation 127
A.3. History of Units 128
B. SQL Key Words 130
Bibliography 146

List of Tables

1.1. Operator Precedence (decreasing) 10
3.1. Data Types 22
3.2. Numeric Types 23
3.3. Monetary Types 27
3.4. Character Types 27
3.5. Specialty Character Type 28
3.6. Binary String Types 28
3.7. SQL Literal Escaped Octets 29
3.8. SQL Output Escaped Octets 29
3.9. Comparison of SQL99 Binary String and PostgreSQL `BYTEA` types 30
3.10. Date/Time Types 30
3.11. Date Input 32
3.12. Time Input 32
3.13. Time With Time Zone Input 33
3.14. Time Zone Input 33
3.15. Special Date/Time Constants 34
3.16. Date/Time Output Styles 35
3.17. Date-Order Conventions 35
3.18. Geometric Types 37
3.19. Network Address Data Types 40
3.20. `cidr` Type Input Examples 41
4.1. Comparison Operators 43
4.2. Mathematical Operators 45
4.3. Bit String Binary Operators 45
4.4. Mathematical Functions 46
4.5. Trigonometric Functions 47
4.6. SQL String Functions and Operators 47
4.7. Other String Functions 48
4.8. SQL Binary String Functions and Operators 50
4.9. Other Binary String Functions 51
4.10. Regular Expression Match Operators 52
4.11. Formatting Functions 55
4.12. Template patterns for date/time conversions 56
4.13. Template pattern modifiers for date/time conversions 57
4.14. Template patterns for numeric conversions 58
4.15. `to_char` Examples 59
4.16. Date/Time Operators 60
4.17. Date/Time Functions 60
4.18. Geometric Operators 66
4.19. Geometric Functions 67
4.20. Geometric Type Conversion Functions 68
4.21. `cidr` and `inet` Operators 69
4.22. `cidr` and `inet` Functions 70
4.23. `macaddr` Functions 70
4.24. Sequence Functions 71
4.25. Session Information Functions 74
4.26. System Information Functions 74
4.27. Access Privilege Inquiry Functions 74
4.28. Catalog Information Functions 75
4.29. Comment Information Functions 75
4.30. Aggregate Functions 76

9.1. Isolation Levels 107
11.1. pg_stats Columns 118
A.1. Month Abbreviations 123
A.2. Day of the Week Abbreviations 123
A.3. Field Modifiers 123
A.4. Time Zones 124
A.5. Australian Time Zones 127
B.1. SQL Key Words 130

List of Examples

3.1. Using the character types 28
3.2. Using the `boolean` type 37
3.3. Using the bit string types 42
5.1. Exponentiation Operator Type Resolution 84
5.2. String Concatenation Operator Type Resolution 85
5.3. Absolute-Value and Factorial Operator Type Resolution 85
5.4. Factorial Function Argument Type Resolution 87
5.5. Substring Function Type Resolution 87
5.6. `character` Storage Type Conversion 89
5.7. Underspecified Types in a Union 89
5.8. Type Conversion in a Simple Union 90
5.9. Type Conversion in a Transposed Union 90
7.1. Setting up a Partial Index to Exclude Common Values 100
7.2. Setting up a Partial Index to Exclude Uninteresting Values 101
7.3. Setting up a Partial Unique Index 101

Preface

1. What is PostgreSQL?

PostgreSQL is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2[1], developed at the University of California at Berkeley Computer Science Department. The POSTGRES project, led by Professor Michael Stonebraker, was sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc.

PostgreSQL is an open-source descendant of this original Berkeley code. It provides SQL92/SQL99 language support and other modern features.

POSTGRES pioneered many of the object-relational concepts now becoming available in some commercial databases. Traditional relational database management systems (RDBMS) support a data model consisting of a collection of named relations, containing attributes of a specific type. In current commercial systems, possible types include floating point numbers, integers, character strings, money, and dates. It is commonly recognized that this model is inadequate for future data-processing applications. The relational model successfully replaced previous models in part because of its "Spartan simplicity". However, this simplicity makes the implementation of certain applications very difficult. PostgreSQL offers substantial additional power by incorporating the following additional concepts in such a way that users can easily extend the system:

- inheritance
- data types
- functions

Other features provide additional power and flexibility:

- constraints
- triggers
- rules
- transactional integrity

These features put PostgreSQL into the category of databases referred to as *object-relational*. Note that this is distinct from those referred to as *object-oriented*, which in general are not as well suited to supporting traditional relational database languages. So, although PostgreSQL has some object-oriented features, it is firmly in the relational database world. In fact, some commercial databases have recently incorporated features pioneered by PostgreSQL.

2. A Short History of PostgreSQL

The object-relational database management system now known as PostgreSQL (and briefly called Postgres95) is derived from the POSTGRES package written at the University of California at Berkeley. With over a decade of development behind it, PostgreSQL is the most advanced open-source database available anywhere, offering multiversion concurrency control, supporting almost all SQL constructs (including subselects, transactions, and user-defined types and functions), and having a wide range of language bindings available (including C, C++, Java, Perl, Tcl, and Python).

[1] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html

2.1. The Berkeley POSTGRES Project

Implementation of the POSTGRES DBMS began in 1986. The initial concepts for the system were presented in [ston86] and the definition of the initial data model appeared in [rowe87]. The design of the rule system at that time was described in [ston87a]. The rationale and architecture of the storage manager were detailed in [ston87b].

Postgres has undergone several major releases since then. The first "demoware" system became operational in 1987 and was shown at the 1988 ACM-SIGMOD Conference. Version 1, described in [ston90a], was released to a few external users in June 1989. In response to a critique of the first rule system ([ston89]), the rule system was redesigned ([ston90b]) and Version 2 was released in June 1990 with the new rule system. Version 3 appeared in 1991 and added support for multiple storage managers, an improved query executor, and a rewritten rewrite rule system. For the most part, subsequent releases until Postgres95 (see below) focused on portability and reliability.

POSTGRES has been used to implement many different research and production applications. These include: a financial data analysis system, a jet engine performance monitoring package, an asteroid tracking database, a medical information database, and several geographic information systems. POSTGRES has also been used as an educational tool at several universities. Finally, Illustra Information Technologies (later merged into Informix[2], which is now owned by IBM[3].) picked up the code and commercialized it. POSTGRES became the primary data manager for the Sequoia 2000[4] scientific computing project in late 1992.

The size of the external user community nearly doubled during 1993. It became increasingly obvious that maintenance of the prototype code and support was taking up large amounts of time that should have been devoted to database research. In an effort to reduce this support burden, the Berkeley POSTGRES project officially ended with Version 4.2.

2.2. Postgres95

In 1994, Andrew Yu and Jolly Chen added a SQL language interpreter to POSTGRES. Postgres95 was subsequently released to the Web to find its own way in the world as an open-source descendant of the original POSTGRES Berkeley code.

Postgres95 code was completely ANSI C and trimmed in size by 25%. Many internal changes improved performance and maintainability. Postgres95 release 1.0.x ran about 30-50% faster on the Wisconsin Benchmark compared to POSTGRES, Version 4.2. Apart from bug fixes, the following were the major enhancements:

- The query language PostQUEL was replaced with SQL (implemented in the server). Subqueries were not supported until PostgreSQL (see below), but they could be imitated in Postgres95 with user-defined SQL functions. Aggregates were re-implemented. Support for the GROUP BY query clause was also added. The `libpq` interface remained available for C programs.
- In addition to the monitor program, a new program (psql) was provided for interactive SQL queries using GNU Readline.
- A new front-end library, `libpgtcl`, supported Tcl-based clients. A sample shell, `pgtclsh`, provided new Tcl commands to interface Tcl programs with the Postgres95 backend.
- The large-object interface was overhauled. The Inversion large objects were the only mechanism for storing large objects. (The Inversion file system was removed.)

[2] http://www.informix.com/
[3] http://www.ibm.com/
[4] http://meteora.ucsd.edu/s2k/s2k_home.html

- The instance-level rule system was removed. Rules were still available as rewrite rules.
- A short tutorial introducing regular SQL features as well as those of Postgres95 was distributed with the source code
- GNU make (instead of BSD make) was used for the build. Also, Postgres95 could be compiled with an unpatched GCC (data alignment of doubles was fixed).

2.3. PostgreSQL

By 1996, it became clear that the name "Postgres95" would not stand the test of time. We chose a new name, PostgreSQL, to reflect the relationship between the original POSTGRES and the more recent versions with SQL capability. At the same time, we set the version numbering to start at 6.0, putting the numbers back into the sequence originally begun by the Berkeley POSTGRES project.

The emphasis during development of Postgres95 was on identifying and understanding existing problems in the backend code. With PostgreSQL, the emphasis has shifted to augmenting features and capabilities, although work continues in all areas.

Major enhancements in PostgreSQL include:

- Table-level locking has been replaced by multiversion concurrency control, which allows readers to continue reading consistent data during writer activity and enables hot backups from pg_dump while the database stays available for queries.
- Important backend features, including subselects, defaults, constraints, and triggers, have been implemented.
- Additional SQL92-compliant language features have been added, including primary keys, quoted identifiers, literal string type coercion, type casting, and binary and hexadecimal integer input.
- Built-in types have been improved, including new wide-range date/time types and additional geometric type support.
- Overall backend code speed has been increased by approximately 20-40%, and backend start-up time has decreased by 80% since version 6.0 was released.

3. Documentation Resources

This manual set is organized into several parts:

Tutorial

An informal introduction for new users

User's Guide

Documents the SQL query language environment, including data types and functions.

Programmer's Guide

Advanced information for application programmers. Topics include type and function extensibility, library interfaces, and application design issues.

Administrator's Guide

Installation and server management information

Reference Manual

Reference pages for SQL command syntax and client and server programs

Developer's Guide

Information for PostgreSQL developers. This is intended for those who are contributing to the PostgreSQL project; application development information appears in the *Programmer's Guide*.

In addition to this manual set, there are other resources to help you with PostgreSQL installation and use:

man pages

The *Reference Manual*'s pages in the traditional Unix man format.

FAQs

Frequently Asked Questions (FAQ) lists document both general issues and some platform-specific issues.

READMEs

README files are available for some contributed packages.

Web Site

The PostgreSQL web site[5] carries details on the latest release, upcoming features, and other information to make your work or play with PostgreSQL more productive.

Mailing Lists

The mailing lists are a good place to have your questions answered, to share experiences with other users, and to contact the developers. Consult the User's Lounge[6] section of the PostgreSQL web site for details.

Yourself!

PostgreSQL is an open-source effort. As such, it depends on the user community for ongoing support. As you begin to use PostgreSQL, you will rely on others for help, either through the documentation or through the mailing lists. Consider contributing your knowledge back. If you learn something which is not in the documentation, write it up and contribute it. If you add features to the code, contribute them.

Even those without a lot of experience can provide corrections and minor changes in the documentation, and that is a good way to start. The <pgsql-docs@postgresql.org> mailing list is the place to get going.

4. Terminology and Notation

The terms “PostgreSQL” and “Postgres” will be used interchangeably to refer to the software that accompanies this documentation.

An *administrator* is generally a person who is in charge of installing and running the server. A *user* could be anyone who is using, or wants to use, any part of the PostgreSQL system. These terms should

[5] http://www.postgresql.org
[6] http://www.postgresql.org/users-lounge/

not be interpreted too narrowly; this documentation set does not have fixed presumptions about system administration procedures.

We use `/usr/local/pgsql/` as the root directory of the installation and `/usr/local/pgsql/data` as the directory with the database files. These directories may vary on your site, details can be derived in the *Administrator's Guide*.

In a command synopsis, brackets (`[` and `]`) indicate an optional phrase or keyword. Anything in braces (`{` and `}`) and containing vertical bars (`|`) indicates that you must choose one alternative.

Examples will show commands executed from various accounts and programs. Commands executed from a Unix shell may be preceded with a dollar sign ("`$`"). Commands executed from particular user accounts such as root or postgres are specially flagged and explained. SQL commands may be preceded with "=>" or will have no leading prompt, depending on the context.

Note

The notation for flagging commands is not universally consistent throughout the documentation set. Please report problems to the documentation mailing list `<pgsql-docs@postgresql.org>`.

5. Bug Reporting Guidelines

When you find a bug in PostgreSQL we want to hear about it. Your bug reports play an important part in making PostgreSQL more reliable because even the utmost care cannot guarantee that every part of PostgreSQL will work on every platform under every circumstance.

The following suggestions are intended to assist you in forming bug reports that can be handled in an effective fashion. No one is required to follow them but it tends to be to everyone's advantage.

We cannot promise to fix every bug right away. If the bug is obvious, critical, or affects a lot of users, chances are good that someone will look into it. It could also happen that we tell you to update to a newer version to see if the bug happens there. Or we might decide that the bug cannot be fixed before some major rewrite we might be planning is done. Or perhaps it is simply too hard and there are more important things on the agenda. If you need help immediately, consider obtaining a commercial support contract.

5.1. Identifying Bugs

Before you report a bug, please read and re-read the documentation to verify that you can really do whatever it is you are trying. If it is not clear from the documentation whether you can do something or not, please report that too; it is a bug in the documentation. If it turns out that the program does something different from what the documentation says, that is a bug. That might include, but is not limited to, the following circumstances:

- A program terminates with a fatal signal or an operating system error message that would point to a problem in the program. (A counterexample might be a "disk full" message, since you have to fix that yourself.)
- A program produces the wrong output for any given input.
- A program refuses to accept valid input (as defined in the documentation).
- A program accepts invalid input without a notice or error message. But keep in mind that your idea of invalid input might be our idea of an extension or compatibility with traditional practice.

- PostgreSQL fails to compile, build, or install according to the instructions on supported platforms.

Here “program” refers to any executable, not only the backend server.

Being slow or resource-hogging is not necessarily a bug. Read the documentation or ask on one of the mailing lists for help in tuning your applications. Failing to comply to the SQL standard is not necessarily a bug either, unless compliance for the specific feature is explicitly claimed.

Before you continue, check on the TODO list and in the FAQ to see if your bug is already known. If you cannot decode the information on the TODO list, report your problem. The least we can do is make the TODO list clearer.

5.2. What to report

The most important thing to remember about bug reporting is to state all the facts and only facts. Do not speculate what you think went wrong, what “it seemed to do”, or which part of the program has a fault. If you are not familiar with the implementation you would probably guess wrong and not help us a bit. And even if you are, educated explanations are a great supplement to but no substitute for facts. If we are going to fix the bug we still have to see it happen for ourselves first. Reporting the bare facts is relatively straightforward (you can probably copy and paste them from the screen) but all too often important details are left out because someone thought it does not matter or the report would be understood anyway.

The following items should be contained in every bug report:

- The exact sequence of steps *from program start-up* necessary to reproduce the problem. This should be self-contained; it is not enough to send in a bare select statement without the preceding create table and insert statements, if the output should depend on the data in the tables. We do not have the time to reverse-engineer your database schema, and if we are supposed to make up our own data we would probably miss the problem. The best format for a test case for query-language related problems is a file that can be run through the psql frontend that shows the problem. (Be sure to not have anything in your `~/.psqlrc` start-up file.) An easy start at this file is to use pg_dump to dump out the table declarations and data needed to set the scene, then add the problem query. You are encouraged to minimize the size of your example, but this is not absolutely necessary. If the bug is reproducible, we will find it either way.

  If your application uses some other client interface, such as PHP, then please try to isolate the offending queries. We will probably not set up a web server to reproduce your problem. In any case remember to provide the exact input files, do not guess that the problem happens for “large files” or “mid-size databases”, etc. since this information is too inexact to be of use.

- The output you got. Please do not say that it “didn't work” or “crashed”. If there is an error message, show it, even if you do not understand it. If the program terminates with an operating system error, say which. If nothing at all happens, say so. Even if the result of your test case is a program crash or otherwise obvious it might not happen on our platform. The easiest thing is to copy the output from the terminal, if possible.

  > **Note**
  >
  > In case of fatal errors, the error message reported by the client might not contain all the information available. Please also look at the log output of the database server. If you do not keep your server's log output, this would be a good time to start doing so.

- The output you expected is very important to state. If you just write “This command gives me that output.” or “This is not what I expected.”, we might run it ourselves, scan the output, and think it looks OK and is exactly what we expected. We should not have to spend the time to decode the exact semantics behind your commands. Especially refrain from merely saying that “This is not what SQL says/Oracle

does.” Digging out the correct behavior from SQL is not a fun undertaking, nor do we all know how all the other relational databases out there behave. (If your problem is a program crash, you can obviously omit this item.)

- Any command line options and other start-up options, including concerned environment variables or configuration files that you changed from the default. Again, be exact. If you are using a prepackaged distribution that starts the database server at boot time, you should try to find out how that is done.

- Anything you did at all differently from the installation instructions.

- The PostgreSQL version. You can run the command `SELECT version();` to find out the version of the server you are connected to. Most executable programs also support a `--version` option; at least `postmaster --version` and `psql --version` should work. If the function or the options do not exist then your version is more than old enough to warrant an upgrade. You can also look into the `README` file in the source directory or at the name of your distribution file or package name. If you run a prepackaged version, such as RPMs, say so, including any subversion the package may have. If you are talking about a CVS snapshot, mention that, including its date and time.

  If your version is older than 7.2.8 we will almost certainly tell you to upgrade. There are tons of bug fixes in each new release, that is why we make new releases.

- Platform information. This includes the kernel name and version, C library, processor, memory information. In most cases it is sufficient to report the vendor and version, but do not assume everyone knows what exactly “Debian” contains or that everyone runs on Pentiums. If you have installation problems then information about compilers, make, etc. is also necessary.

Do not be afraid if your bug report becomes rather lengthy. That is a fact of life. It is better to report everything the first time than us having to squeeze the facts out of you. On the other hand, if your input files are huge, it is fair to ask first whether somebody is interested in looking into it.

Do not spend all your time to figure out which changes in the input make the problem go away. This will probably not help solving it. If it turns out that the bug cannot be fixed right away, you will still have time to find and share your work-around. Also, once again, do not waste your time guessing why the bug exists. We will find that out soon enough.

When writing a bug report, please choose non-confusing terminology. The software package in total is called “PostgreSQL”, sometimes “Postgres” for short. If you are specifically talking about the backend server, mention that, do not just say “PostgreSQL crashes”. A crash of a single backend server process is quite different from crash of the parent “postmaster” process; please don't say “the postmaster crashed” when you mean a single backend went down, nor vice versa. Also, client programs such as the interactive frontend “psql” are completely separate from the backend. Please try to be specific about whether the problem is on the client or server side.

5.3. Where to report bugs

In general, send bug reports to the bug report mailing list at <pgsql-bugs@postgresql.org>. You are requested to use a descriptive subject for your email message, perhaps parts of the error message.

Another method is to fill in the bug report web-form available at the project's web site http://www.postgresql.org/. Entering a bug report this way causes it to be mailed to the <pgsql-bugs@postgresql.org> mailing list.

If your bug report has security implications and you'd prefer that it not become immediately visible in public archives, don't send it to `pgsql-bugs`. Security issues can be reported privately to <security@postgresql.org>.

Do not send bug reports to any of the user mailing lists, such as <pgsql-sql@postgresql.org> or <pgsql-general@postgresql.org>. These mailing lists are for answering user questions and their subscribers normally do not wish to receive bug reports. More importantly, they are unlikely to fix them.

Also, please do *not* send reports to the developers' mailing list <pgsql-hackers@postgresql.org>. This list is for discussing the development of PostgreSQL and it would be nice if we could keep the bug reports separate. We might choose to take up a discussion about your bug report on pgsql-hackers, if the problem needs more review.

If you have a problem with the documentation, the best place to report it is the documentation mailing list <pgsql-docs@postgresql.org>. Please be specific about what part of the documentation you are unhappy with.

If your bug is a portability problem on a non-supported platform, send mail to <pgsql-ports@postgresql.org>, so we (and you) can work on porting PostgreSQL to your platform.

Note

Due to the unfortunate amount of spam going around, all of the above email addresses are closed mailing lists. That is, you need to be subscribed to a list to be allowed to post on it. (You need not be subscribed to use the bug report web-form, however.) If you would like to send mail but do not want to receive list traffic, you can subscribe and set your subscription option to nomail. For more information send mail to <majordomo@postgresql.org> with the single word help in the body of the message.

6. Y2K Statement

Author

Written by Thomas Lockhart (<lockhart@fourpalms.org>) on 1998-10-22. Updated 2000-03-31.

The PostgreSQL Global Development Group provides the PostgreSQL software code tree as a public service, without warranty and without liability for its behavior or performance. However, at the time of writing:

- The author of this statement, a volunteer on the PostgreSQL support team since November, 1996, is not aware of any problems in the PostgreSQL code base related to time transitions around Jan 1, 2000 (Y2K).
- The author of this statement is not aware of any reports of Y2K problems uncovered in regression testing or in other field use of recent or current versions of PostgreSQL. We might have expected to hear about problems if they existed, given the installed base and the active participation of users on the support mailing lists.
- To the best of the author's knowledge, the assumptions PostgreSQL makes about dates specified with a two-digit year are documented in the current *User's Guide* in the chapter on data types. For two-digit years, the significant transition year is 1970, not 2000; e.g. 70-01-01 is interpreted as 1970-01-01, whereas 69-01-01 is interpreted as 2069-01-01.
- Any Y2K problems in the underlying OS related to obtaining the “current time” may propagate into apparent Y2K problems in PostgreSQL.

Refer to The GNU Project[7] and The Perl Institute[8] for further discussion of Y2K issues, particularly as it relates to open source, no fee software.

[7] http://www.gnu.org/software/year2000.html
[8] http://language.perl.com/news/y2k.html

Chapter 1. SQL Syntax

This chapter describes the syntax of SQL.

1.1. Lexical Structure

SQL input consists of a sequence of *commands*. A command is composed of a sequence of *tokens*, terminated by a semicolon (";"). The end of the input stream also terminates a command. Which tokens are valid depends on the syntax of the particular command.

A token can be a *key word*, an *identifier*, a *quoted identifier*, a *literal* (or constant), or a special character symbol. Tokens are normally separated by whitespace (space, tab, newline), but need not be if there is no ambiguity (which is generally only the case if a special character is adjacent to some other token type).

Additionally, *comments* can occur in SQL input. They are not tokens, they are effectively equivalent to whitespace.

For example, the following is (syntactically) valid SQL input:

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

This is a sequence of three commands, one per line (although this is not required; more than one command can be on a line, and commands can usefully be split across lines).

The SQL syntax is not very consistent regarding what tokens identify commands and which are operands or parameters. The first few tokens are generally the command name, so in the above example we would usually speak of a "SELECT", an "UPDATE", and an "INSERT" command. But for instance the `UPDATE` command always requires a `SET` token to appear in a certain position, and this particular variation of `INSERT` also requires a `VALUES` in order to be complete. The precise syntax rules for each command are described in the *Reference Manual*.

1.1.1. Identifiers and Key Words

Tokens such as `SELECT`, `UPDATE`, or `VALUES` in the example above are examples of *key words*, that is, words that have a fixed meaning in the SQL language. The tokens `MY_TABLE` and `A` are examples of *identifiers*. They identify names of tables, columns, or other database objects, depending on the command they are used in. Therefore they are sometimes simply called "names". Key words and identifiers have the same lexical structure, meaning that one cannot know whether a token is an identifier or a key word without knowing the language. A complete list of key words can be found in Appendix B, *SQL Key Words*.

SQL identifiers and key words must begin with a letter (`a-z`, but also letters with diacritical marks and non-Latin letters) or an underscore (_). Subsequent characters in an identifier or key word can be letters, digits (`0-9`), or underscores, although the SQL standard will not define a key word that contains digits or starts or ends with an underscore.

The system uses no more than `NAMEDATALEN`-1 characters of an identifier; longer names can be written in commands, but they will be truncated. By default, `NAMEDATALEN` is 32 so the maximum identifier length is 31 (but at the time the system is built, `NAMEDATALEN` can be changed in `src/include/postgres_ext.h`).

Identifier and key word names are case insensitive. Therefore

```
UPDATE MY_TABLE SET A = 5;
```

can equivalently be written as

```
uPDaTE my_TabLE SeT a = 5;
```

A convention often used is to write key words in upper case and names in lower case, e.g.,

```
UPDATE my_table SET a = 5;
```

There is a second kind of identifier: the *delimited identifier* or *quoted identifier*. It is formed by enclosing an arbitrary sequence of characters in double-quotes (`"`). A delimited identifier is always an identifier, never a key word. So `"select"` could be used to refer to a column or table named "select", whereas an unquoted `select` would be taken as a key word and would therefore provoke a parse error when used where a table or column name is expected. The example can be written with quoted identifiers like this:

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

Quoted identifiers can contain any character other than a double quote itself. This allows constructing table or column names that would otherwise not be possible, such as ones containing spaces or ampersands. The length limitation still applies.

Quoting an identifier also makes it case-sensitive, whereas unquoted names are always folded to lower case. For example, the identifiers `FOO`, `foo` and `"foo"` are considered the same by PostgreSQL, but `"Foo"` and `"FOO"` are different from these three and each other. [1]

1.1.2. Constants

There are four kinds of *implicitly-typed constants* in PostgreSQL: strings, bit strings, integers, and floating-point numbers. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the system. The implicit constants are described below; explicit constants are discussed afterwards.

1.1.2.1. String Constants

A string constant in SQL is an arbitrary sequence of characters bounded by single quotes ("'"), e.g., `'This is a string'`. SQL allows single quotes to be embedded in strings by typing two adjacent single quotes (e.g., `'Dianne''s horse'`). In PostgreSQL single quotes may alternatively be escaped with a backslash ("\", e.g., `'Dianne\'s horse'`).

C-style backslash escapes are also available: `\b` is a backspace, `\f` is a form feed, `\n` is a newline, `\r` is a carriage return, `\t` is a tab, and `\`*xxx*, where *xxx* is an octal number, is the character with the corresponding ASCII code. Any other character following a backslash is taken literally. Thus, to include a backslash in a string constant, type two backslashes.

The character with the code zero cannot be in a string constant.

Two string constants that are only separated by whitespace *with at least one newline* are concatenated and effectively treated as if the string had been written in one constant. For example:

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

[1] The folding of unquoted names to lower case in PostgreSQL is incompatible with the SQL standard, which says that unquoted names should be folded to upper case. Thus, `foo` should be equivalent to `"FOO"` not `"foo"` according to the standard. If you want to write portable applications you are advised to always quote a particular name or never quote it.

is equivalent to

```
SELECT 'foobar';
```

but

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

is not valid syntax, and PostgreSQL is consistent with SQL9x in this regard.

1.1.2.2. Bit-String Constants

Bit-string constants look like string constants with a `B` (upper or lower case) immediately before the opening quote (no intervening whitespace), e.g., `B'1001'`. The only characters allowed within bit-string constants are `0` and `1`. Bit-string constants can be continued across lines in the same way as regular string constants.

1.1.2.3. Integer Constants

Integer constants in SQL are sequences of decimal digits (0 though 9) with no decimal point and no exponent. The range of legal values depends on which integer data type is used, but the plain `integer` type accepts values ranging from -2147483648 to +2147483647. (The optional plus or minus sign is actually a separate unary operator and not part of the integer constant.)

1.1.2.4. Floating-Point Constants

Floating-point constants are accepted in these general forms:

digits.[*digits*][e[+-]*digits*]
[*digits*].*digits*[e[+-]*digits*]
*digits*e[+-]*digits*

where *digits* is one or more decimal digits. At least one digit must be before or after the decimal point. At least one digit must follow the exponent delimiter (`e`) if that field is present. Thus, a floating-point constant is distinguished from an integer constant by the presence of either the decimal point or the exponent clause (or both). There must not be a space or other characters embedded in the constant.

These are some examples of valid floating-point constants:

3.5
4.
.001
5e2
1.925e-3

Floating-point constants are of type `DOUBLE PRECISION`. `REAL` can be specified explicitly by using SQL string notation or PostgreSQL type notation:

```
REAL '1.23'  -- string style
'1.23'::REAL -- PostgreSQL (historical) style
```

1.1.2.5. Constants of Other Types

A constant of an *arbitrary* type can be entered using any one of the following notations:

```
type 'string'
'string'::type
CAST ( 'string' AS type )
```

The string's text is passed to the input conversion routine for the type called *type*. The result is a constant of the indicated type. The explicit type cast may be omitted if there is no ambiguity as to the type the constant must be (for example, when it is passed as an argument to a non-overloaded function), in which case it is automatically coerced.

It is also possible to specify a type coercion using a function-like syntax:

```
typename ( 'string' )
```

but not all type names may be used in this way; see Section 1.3.6, “Type Casts” for details.

The `::`, `CAST()`, and function-call syntaxes can also be used to specify runtime type conversions of arbitrary expressions, as discussed in Section 1.3.6, “Type Casts”. But the form *type* '*string*' can only be used to specify the type of a literal constant. Another restriction on *type* '*string*' is that it does not work for array types; use `::` or `CAST()` to specify the type of an array constant.

1.1.2.6. Array constants

The general format of an array constant is the following:

```
'{ val1 delim val2 delim ... }'
```

where *delim* is the delimiter character for the type, as recorded in its `pg_type` entry. (For all built-in types, this is the comma character “`,`”.) Each *val* is either a constant of the array element type, or a subarray. An example of an array constant is

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

This constant is a two-dimensional, 3-by-3 array consisting of three subarrays of integers.

Individual array elements can be placed between double-quote marks (`"`) to avoid ambiguity problems with respect to whitespace. Without quote marks, the array-value parser will skip leading whitespace.

(Array constants are actually only a special case of the generic type constants discussed in the previous section. The constant is initially treated as a string and passed to the array input conversion routine. An explicit type specification might be necessary.)

1.1.3. Operators

An operator is a sequence of up to `NAMEDATALEN`-1 (31 by default) characters from the following list:

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

There are a few restrictions on operator names, however:

- `$` (dollar) cannot be a single-character operator, although it can be part of a multiple-character operator name.
- `--` and `/*` cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
- A multiple-character operator name cannot end in + or −, unless the name also contains at least one of these characters:

~ ! @ # % ^ & | ` ? $

For example, `@-` is an allowed operator name, but `*-` is not. This restriction allows PostgreSQL to parse SQL-compliant queries without requiring spaces between tokens.

When working with non-SQL-standard operator names, you will usually need to separate adjacent operators with spaces to avoid ambiguity. For example, if you have defined a left unary operator named `@`, you cannot write `X*@Y`; you must write `X* @Y` to ensure that PostgreSQL reads it as two operator names not one.

1.1.4. Special Characters

Some characters that are not alphanumeric have a special meaning that is different from being an operator. Details on the usage can be found at the location where the respective syntax element is described. This section only exists to advise the existence and summarize the purposes of these characters.

- A dollar sign (`$`) followed by digits is used to represent the positional parameters in the body of a function definition. In other contexts the dollar sign may be part of an operator name.
- Parentheses (`()`) have their usual meaning to group expressions and enforce precedence. In some cases parentheses are required as part of the fixed syntax of a particular SQL command.
- Brackets (`[]`) are used to select the elements of an array. See Chapter 6, *Arrays* for more information on arrays.
- Commas (`,`) are used in some syntactical constructs to separate the elements of a list.
- The semicolon (`;`) terminates an SQL command. It cannot appear anywhere within a command, except within a string constant or quoted identifier.
- The colon (`:`) is used to select "slices" from arrays. (See Chapter 6, *Arrays*.) In certain SQL dialects (such as Embedded SQL), the colon is used to prefix variable names.
- The asterisk (`*`) has a special meaning when used in the `SELECT` command or with the `COUNT` aggregate function.
- The period (`.`) is used in floating-point constants, and to separate table and column names.

1.1.5. Comments

A comment is an arbitrary sequence of characters beginning with double dashes and extending to the end of the line, e.g.:

```
-- This is a standard SQL92 comment
```

Alternatively, C-style block comments can be used:

```
/* multiline comment
 * with nesting: /* nested block comment */
 */
```

where the comment begins with `/*` and extends to the matching occurrence of `*/`. These block comments nest, as specified in SQL99 but unlike C, so that one can comment out larger blocks of code that may contain existing block comments.

A comment is removed from the input stream before further syntax analysis and is effectively replaced by whitespace.

1.2. Columns

A *column* is either a user-defined column of a given table or one of the following system-defined columns:

`oid`

The object identifier (object ID) of a row. This is a serial number that is automatically added by PostgreSQL to all table rows (unless the table was created WITHOUT OIDS, in which case this column is not present).

`tableoid`

The OID of the table containing this row. This attribute is particularly handy for queries that select from inheritance hierarchies, since without it, it's difficult to tell which individual table a row came from. The `tableoid` can be joined against the `oid` column of `pg_class` to obtain the table name.

`xmin`

The identity (transaction ID) of the inserting transaction for this tuple. (Note: A tuple is an individual state of a row; each update of a row creates a new tuple for the same logical row.)

`cmin`

The command identifier (starting at zero) within the inserting transaction.

`xmax`

The identity (transaction ID) of the deleting transaction, or zero for an undeleted tuple. It is possible for this field to be nonzero in a visible tuple: that usually indicates that the deleting transaction hasn't committed yet, or that an attempted deletion was rolled back.

`cmax`

The command identifier within the deleting transaction, or zero.

`ctid`

The tuple ID of the tuple within its table. This is a pair (block number, tuple index within block) that identifies the physical location of the tuple. Note that although the `ctid` can be used to locate the tuple very quickly, a row's `ctid` will change each time it is updated or moved by `VACUUM FULL`. Therefore `ctid` is useless as a long-term row identifier. The OID, or even better a user-defined serial number, should be used to identify logical rows.

OIDs are 32-bit quantities and are assigned from a single cluster-wide counter. In a large or long-lived database, it is possible for the counter to wrap around. Hence, it is bad practice to assume that OIDs are unique, unless you take steps to ensure that they are unique. Recommended practice when using OIDs for row identification is to create a unique constraint on the OID column of each table for which the OID will be used. Never assume that OIDs are unique across tables; use the combination of `tableoid` and row OID if you need a database-wide identifier. (Future releases of PostgreSQL are likely to use a separate OID counter for each table, so that `tableoid` *must* be included to arrive at a globally unique identifier.)

Transaction identifiers are 32-bit quantities. In a long-lived database it is possible for transaction IDs to wrap around. This is not a fatal problem given appropriate maintenance procedures; see the *Administrator's*

Guide for details. However, it is unwise to depend on uniqueness of transaction IDs over the long term (more than one billion transactions).

Command identifiers are also 32-bit quantities. This creates a hard limit of 2^{32} (4 billion) SQL commands within a single transaction. In practice this limit is not a problem --- note that the limit is on number of SQL queries, not number of tuples processed.

1.3. Value Expressions

Value expressions are used in a variety of contexts, such as in the target list of the `SELECT` command, as new column values in `INSERT` or `UPDATE`, or in search conditions in a number of commands. The result of a value expression is sometimes called a *scalar*, to distinguish it from the result of a table expression (which is a table). Value expressions are therefore also called *scalar expressions* (or even simply *expressions*). The expression syntax allows the calculation of values from primitive parts using arithmetic, logical, set, and other operations.

A value expression is one of the following:

- A constant or literal value; see Section 1.1.2, “Constants”.
- A column reference.
- A positional parameter reference, in the body of a function declaration.
- An operator invocation.
- A function call.
- An aggregate expression.
- A type cast.
- A scalar subquery.
- `( expression )`

  Parentheses are used to group subexpressions and override precedence.

In addition to this list, there are a number of constructs that can be classified as an expression but do not follow any general syntax rules. These generally have the semantics of a function or operator and are explained in the appropriate location in Chapter 4, *Functions and Operators*. An example is the `IS NULL` clause.

We have already discussed constants in Section 1.1.2, “Constants”. The following sections discuss the remaining options.

1.3.1. Column References

A column can be referenced in the form:

correlation.*columnname* `['subscript`]'

correlation is either the name of a table, an alias for a table defined by means of a FROM clause, or the key words `NEW` or `OLD`. (NEW and OLD can only appear in the action portion of a rule, while other correlation names can be used in any SQL statement.) The correlation name and separating dot may be omitted if the column name is unique across all the tables being used in the current query. If *column* is

of an array type, then the optional *subscript* selects a specific element or elements in the array. If no subscript is provided, then the whole array is selected. (See Chapter 6, *Arrays* for more about arrays.)

1.3.2. Positional Parameters

A positional parameter reference is used to indicate a parameter in an SQL function. Typically this is used in SQL function definition statements. The form of a parameter is:

$number

For example, consider the definition of a function, dept, as

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

Here the $1 will be replaced by the first function argument when the function is invoked.

1.3.3. Operator Invocations

There are three possible syntaxes for an operator invocation:

expression operator expression (binary infix operator)
operator expression (unary prefix operator)
expression operator (unary postfix operator)
where the *operator* token follows the syntax rules of Section 1.1.3, "Operators" or is one of the tokens AND, OR, and NOT. Which particular operators exist and whether they are unary or binary depends on what operators have been defined by the system or the user. Chapter 4, *Functions and Operators* describes the built-in operators.

1.3.4. Function Calls

The syntax for a function call is the name of a function (which is subject to the syntax rules for identifiers of Section 1.1.1, "Identifiers and Key Words"), followed by its argument list enclosed in parentheses:

function ([*expression* [, *expression* ...]])

For example, the following computes the square root of 2:

```
sqrt(2)
```

The list of built-in functions is in Chapter 4, *Functions and Operators*. Other functions may be added by the user.

1.3.5. Aggregate Expressions

An *aggregate expression* represents the application of an aggregate function across the rows selected by a query. An aggregate function reduces multiple inputs to a single output value, such as the sum or average of the inputs. The syntax of an aggregate expression is one of the following:

aggregate_name (*expression*)
aggregate_name (ALL *expression*)
aggregate_name (DISTINCT *expression*)
aggregate_name (*)

where *aggregate_name* is a previously defined aggregate, and *expression* is any value expression that does not itself contain an aggregate expression.

The first form of aggregate expression invokes the aggregate across all input rows for which the given expression yields a non-NULL value. (Actually, it is up to the aggregate function whether to ignore NULLs or not --- but all the standard ones do.) The second form is the same as the first, since `ALL` is the default. The third form invokes the aggregate for all distinct non-NULL values of the expression found in the input rows. The last form invokes the aggregate once for each input row regardless of NULL or non-NULL values; since no particular input value is specified, it is generally only useful for the `count()` aggregate function.

For example, `count(*)` yields the total number of input rows; `count(f1)` yields the number of input rows in which `f1` is non-NULL; `count(distinct f1)` yields the number of distinct non-NULL values of `f1`.

The predefined aggregate functions are described in Section 4.14, “Aggregate Functions”. Other aggregate functions may be added by the user.

1.3.6. Type Casts

A type cast specifies a conversion from one data type to another. PostgreSQL accepts two equivalent syntaxes for type casts:

```
CAST ( expression AS type )
expression::type
```

The `CAST` syntax conforms to SQL92; the syntax with `::` is historical PostgreSQL usage.

When a cast is applied to a value expression of a known type, it represents a run-time type conversion. The cast will succeed only if a suitable type conversion function is available. Notice that this is subtly different from the use of casts with constants, as shown in Section 1.1.2.5, “Constants of Other Types”. A cast applied to an unadorned string literal represents the initial assignment of a type to a literal constant value, and so it will succeed for any type (if the contents of the string literal are acceptable input syntax for the data type).

An explicit type cast may be omitted if there is no ambiguity as to the type that a value expression must produce (for example, when it is assigned to a table column); the system will automatically apply a type cast in such cases.

It is also possible to specify a type cast using a function-like syntax:

```
typename ( expression )
```

However, this only works for types whose names are also valid as function names. For example, `double precision` can't be used this way, but the equivalent `float8` can. Also, the names `interval`, `time`, and `timestamp` can only be used in this fashion if they are double-quoted, because of parser conflicts. Therefore, the use of the function-like cast syntax leads to inconsistencies and should probably be avoided in new applications.

1.3.7. Scalar Subqueries

A scalar subquery is an ordinary `SELECT` in parentheses that returns exactly one row with one column. The `SELECT` query is executed and the single returned value is used in the surrounding value expression. It is an error to use a query that returns more than one row or more than one column as a scalar subquery. (But if, during a particular execution, the subquery returns no rows, there is no error; the scalar result is

taken to be NULL.) The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery. See also Section 4.15, “Subquery Expressions”.

For example, the following finds the largest city population in each state:

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state =
 states.name)
FROM states;
```

1.4. Lexical Precedence

The precedence and associativity of the operators is hard-wired into the parser. Most operators have the same precedence and are left-associative. This may lead to non-intuitive behavior; for example the Boolean operators < and > have a different precedence than the Boolean operators <= and >=. Also, you will sometimes need to add parentheses when using combinations of binary and unary operators. For instance

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

will be parsed as

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

because the parser has no idea -- until it is too late -- that `!` is defined as a postfix operator, not an infix one. To get the desired behavior in this case, you must write

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

This is the price one pays for extensibility.

Table 1.1. Operator Precedence (decreasing)

Operator/Element	Associativity	Description
`::`	left	PostgreSQL-style typecast
`[ ]`	left	array element selection
`.`	left	table/column name separator
–	right	unary minus
`^`	left	exponentiation
`* / %`	left	multiplication, division, modulo
`+ –`	left	addition, subtraction
`IS`		test for TRUE, FALSE, UNKNOWN, NULL
`ISNULL`		test for NULL
`NOTNULL`		test for NOT NULL
(any other)	left	all other native and user-defined operators
`IN`		set membership
`BETWEEN`		containment
`OVERLAPS`		time interval overlap
`LIKE ILIKE`		string pattern matching

Operator/Element	Associativity	Description
< >		less than, greater than
=	right	equality, assignment
NOT	right	logical negation
AND	left	logical conjunction
OR	left	logical disjunction

Note that the operator precedence rules also apply to user-defined operators that have the same names as the built-in operators mentioned above. For example, if you define a “+” operator for some custom data type it will have the same precedence as the built-in “+” operator, no matter what yours does.

Chapter 2. Queries

2.1. Overview

A *query* is the process of retrieving or the command to retrieve data from a database. In SQL the `SELECT` command is used to specify queries. The general syntax of the `SELECT` command is

```
SELECT select_list FROM table_expression [sort_specification]
```

The following sections describe the details of the select list, the table expression, and the sort specification. The simplest kind of query has the form

```
SELECT * FROM table1;
```

Assuming that there is a table called table1, this command would retrieve all rows and all columns from table1. (The method of retrieval depends on the client application. For example, the psql program will display an ASCII-art table on the screen, client libraries will offer functions to retrieve individual rows and columns.) The select list specification `*` means all columns that the table expression happens to provide. A select list can also select a subset of the available columns or even make calculations on the columns before retrieving them; see Section 2.3, “Select Lists”. For example, if table1 has columns named a, b, and c (and perhaps others) you can make the following query:

```
SELECT a, b + c FROM table1;
```

(assuming that b and c are of a numeric data type).

`FROM table1` is a particularly simple kind of table expression. In general, table expressions can be complex constructs of base tables, joins, and subqueries. But you can also omit the table expression entirely and use the SELECT command as a calculator:

```
SELECT 3 * 4;
```

This is more useful if the expressions in the select list return varying results. For example, you could call a function this way.

```
SELECT random();
```

2.2. Table Expressions

A *table expression* specifies a table. The table expression contains a FROM clause that is optionally followed by WHERE, GROUP BY, and HAVING clauses. Trivial table expressions simply refer to a table on disk, a so-called base table, but more complex expressions can be used to modify or combine base tables in various ways.

The optional WHERE, GROUP BY, and HAVING clauses in the table expression specify a pipeline of successive transformations performed on the table derived in the FROM clause. The derived table that is produced by all these transformations provides the input rows used to compute output rows as specified by the select list of column value expressions.

2.2.1. FROM clause

The FROM clause derives a table from one or more other tables given in a comma-separated table reference list.

```
FROM table_reference [, table_reference [, ...]]
```

A table reference may be a table name or a derived table such as a subquery, a table join, or complex combinations of these. If more than one table reference is listed in the FROM clause they are cross-joined (see below) to form the derived table that may then be subject to transformations by the WHERE, GROUP BY, and HAVING clauses and is finally the result of the overall table expression.

When a table reference names a table that is the supertable of a table inheritance hierarchy, the table reference produces rows of not only that table but all of its subtable successors, unless the keyword ONLY precedes the table name. However, the reference produces only the columns that appear in the named table --- any columns added in subtables are ignored.

2.2.1.1. Joined Tables

A joined table is a table derived from two other (real or derived) tables according to the rules of the particular join type. INNER, OUTER, and CROSS JOIN are supported.

Join Types

CROSS JOIN

```
T1 CROSS JOIN T2
```

For each combination of rows from *T1* and *T2*, the derived table will contain a row consisting of all columns in *T1* followed by all columns in *T2*. If the tables have N and M rows respectively, the joined table will have N * M rows. A cross join is equivalent to an `INNER JOIN ON TRUE`.

Tip

`FROM T1 CROSS JOIN T2` is equivalent to `FROM T1, T2`.

Qualified joins

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
 ON boolean_expression
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING
 ( join column list )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

The words `INNER` and `OUTER` are optional for all joins. `INNER` is the default; `LEFT`, `RIGHT`, and `FULL` imply an OUTER JOIN.

The *join condition* is specified in the ON or USING clause, or implicitly by the word NATURAL. The join condition determines which rows from the two source tables are considered to “match”, as explained in detail below.

The ON clause is the most general kind of join condition: it takes a Boolean value expression of the same kind as is used in a WHERE clause. A pair of rows from T1 and T2 match if the ON expression evaluates to TRUE for them.

USING is a shorthand notation: it takes a comma-separated list of column names, which the joined tables must have in common, and forms a join condition specifying equality of each of these pairs of columns. Furthermore, the output of a JOIN USING has one column for each of the equated pairs of input columns, followed by all of the other columns from each table. Thus, `USING (a, b, c)` is equivalent to `ON (t1.a = t2.a AND t1.b = t2.b AND t1.c = t2.c)` with the

exception that if ON is used there will be two columns a, b, and c in the result, whereas with USING there will be only one of each.

Finally, NATURAL is a shorthand form of USING: it forms a USING list consisting of exactly those column names that appear in both input tables. As with USING, these columns appear only once in the output table.

The possible types of qualified JOIN are:

INNER JOIN

For each row R1 of T1, the joined table has a row for each row in T2 that satisfies the join condition with R1.

LEFT OUTER JOIN

First, an INNER JOIN is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is returned with NULL values in columns of T2. Thus, the joined table unconditionally has at least one row for each row in T1.

RIGHT OUTER JOIN

First, an INNER JOIN is performed. Then, for each row in T2 that does not satisfy the join condition with any row in T1, a joined row is returned with NULL values in columns of T1. This is the converse of a left join: the result table will unconditionally have a row for each row in T2.

FULL OUTER JOIN

First, an INNER JOIN is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is returned with null values in columns of T2. Also, for each row of T2 that does not satisfy the join condition with any row in T1, a joined row with null values in the columns of T1 is returned.

Joins of all types can be chained together or nested: either or both of *T1* and *T2* may be joined tables. Parentheses may be used around JOIN clauses to control the join order. In the absence of parentheses, JOIN clauses nest left-to-right.

2.2.1.2. Subqueries

Subqueries specifying a derived table must be enclosed in parentheses and *must* be named using an AS clause. (See Section 2.2.1.3, “Table and Column Aliases”.)

```
FROM (SELECT * FROM table1) AS alias_name
```

This example is equivalent to `FROM table1 AS alias_name`. More interesting cases, which can't be reduced to a plain join, arise when the subquery involves grouping or aggregation.

2.2.1.3. Table and Column Aliases

A temporary name can be given to tables and complex table references to be used for references to the derived table in further processing. This is called a *table alias*.

```
FROM table_reference AS alias
```

Here, *alias* can be any regular identifier. The alias becomes the new name of the table reference for the current query -- it is no longer possible to refer to the table by the original name. Thus

```
SELECT * FROM my_table AS m WHERE my_table.a > 5;
```

is not valid SQL syntax. What will actually happen (this is a PostgreSQL extension to the standard) is that an implicit table reference is added to the FROM clause, so the query is processed as if it were written as

```
SELECT * FROM my_table AS m, my_table AS my_table WHERE my_table.a >
 5;
```

Table aliases are mainly for notational convenience, but it is necessary to use them when joining a table to itself, e.g.,

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
```

Additionally, an alias is required if the table reference is a subquery.

Parentheses are used to resolve ambiguities. The following statement will assign the alias `b` to the result of the join, unlike the previous example:

```
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

```
FROM table_reference alias
```

This form is equivalent to the previously treated one; the `AS` key word is noise.

```
FROM table_reference [AS] alias ( column1 [, column2 [, ...]] )
```

In this form, in addition to renaming the table as described above, the columns of the table are also given temporary names for use by the surrounding query. If fewer column aliases are specified than the actual table has columns, the remaining columns are not renamed. This syntax is especially useful for self-joins or subqueries.

When an alias is applied to the output of a JOIN clause, using any of these forms, the alias hides the original names within the JOIN. For example,

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

is valid SQL, but

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

is not valid: the table alias A is not visible outside the alias C.

2.2.1.4. Examples

```
FROM T1 INNER JOIN T2 USING (C)
FROM T1 LEFT OUTER JOIN T2 USING (C)
FROM (T1 RIGHT OUTER JOIN T2 ON (T1.C1=T2.C1)) AS DT1
FROM (T1 FULL OUTER JOIN T2 USING (C)) AS DT1 (DT1C1, DT1C2)

FROM T1 NATURAL INNER JOIN T2
FROM T1 NATURAL LEFT OUTER JOIN T2
FROM T1 NATURAL RIGHT OUTER JOIN T2
FROM T1 NATURAL FULL OUTER JOIN T2

FROM (SELECT * FROM T1) DT1 CROSS JOIN T2, T3
FROM (SELECT * FROM T1) DT1, T2, T3
```

Above are some examples of joined tables and complex derived tables. Notice how the AS clause renames or names a derived table and how the optional comma-separated list of column names that follows renames the columns. The last two FROM clauses produce the same derived table from T1, T2, and T3. The AS keyword was omitted in naming the subquery as DT1. The keywords OUTER and INNER are noise that can be omitted also.

2.2.2. WHERE clause

The syntax of the WHERE clause is

`WHERE` *`search_condition`*

where *`search_condition`* is any value expression as defined in Section 1.3, “Value Expressions” that returns a value of type `boolean`.

After the processing of the FROM clause is done, each row of the derived table is checked against the search condition. If the result of the condition is true, the row is kept in the output table, otherwise (that is, if the result is false or NULL) it is discarded. The search condition typically references at least some column in the table generated in the FROM clause; this is not required, but otherwise the WHERE clause will be fairly useless.

Note

Before the implementation of the JOIN syntax, it was necessary to put the join condition of an inner join in the WHERE clause. For example, these table expressions are equivalent:

```
FROM a, b WHERE a.id = b.id AND b.val > 5
```

and

```
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
```

or perhaps even

```
FROM a NATURAL JOIN b WHERE b.val > 5
```

Which one of these you use is mainly a matter of style. The JOIN syntax in the FROM clause is probably not as portable to other products. For outer joins there is no choice in any case: they must be done in the FROM clause. A ON/USING clause of an outer join is *not* equivalent to a WHERE condition, because it determines the addition of rows (for unmatched input rows) as well as the removal of rows from the final result.

```
FROM FDT WHERE
    C1 > 5

FROM FDT WHERE
    C1 IN (1, 2, 3)
FROM FDT WHERE
    C1 IN (SELECT C1 FROM T2)
FROM FDT WHERE
    C1 IN (SELECT C3 FROM T2 WHERE C2 = FDT.C1 + 10)

FROM FDT WHERE
    C1 BETWEEN (SELECT C3 FROM T2 WHERE C2 = FDT.C1 + 10) AND 100

FROM FDT WHERE
```

```
    EXISTS (SELECT C1 FROM T2 WHERE C2 > FDT.C1)
```

In the examples above, `FDT` is the table derived in the FROM clause. Rows that do not meet the search condition of the where clause are eliminated from `FDT`. Notice the use of scalar subqueries as value expressions. Just like any other query, the subqueries can employ complex table expressions. Notice how `FDT` is referenced in the subqueries. Qualifying `C1` as `FDT.C1` is only necessary if `C1` is also the name of a column in the derived input table of the subquery. Qualifying the column name adds clarity even when it is not needed. This shows how the column naming scope of an outer query extends into its inner queries.

2.2.3. GROUP BY and HAVING clauses

After passing the WHERE filter, the derived input table may be subject to grouping, using the GROUP BY clause, and elimination of group rows using the HAVING clause.

```
SELECT select_list
    FROM ...
    [WHERE ...]
    GROUP BY grouping_column_reference
 [, grouping_column_reference]...
```

The GROUP BY clause is used to group together rows in a table that share the same values in all the columns listed. The order in which the columns are listed does not matter (as opposed to an ORDER BY clause). The purpose is to reduce each group of rows sharing common values into one group row that is representative of all rows in the group. This is done to eliminate redundancy in the output and/or obtain aggregates that apply to these groups.

Once a table is grouped, columns that are not used in the grouping cannot be referenced except in aggregate expressions, since a specific value in those columns is ambiguous - which row in the group should it come from? The grouped-by columns can be referenced in select list column expressions since they have a known constant value per group. Aggregate functions on the ungrouped columns provide values that span the rows of a group, not of the whole table. For instance, a `sum(sales)` on a table grouped by product code gives the total sales for each product, not the total sales on all products. Aggregates computed on the ungrouped columns are representative of the group, whereas individual values of an ungrouped column are not.

Example:

```
SELECT pid, p.name, (sum(s.units) * p.price) AS sales
  FROM products p LEFT JOIN sales s USING ( pid )
  GROUP BY pid, p.name, p.price;
```

In this example, the columns `pid`, `p.name`, and `p.price` must be in the GROUP BY clause since they are referenced in the query select list. The column s.units does not have to be in the GROUP BY list since it is only used in an aggregate expression (`sum()`), which represents the group of sales of a product. For each product, a summary row is returned about all sales of the product.

In strict SQL, GROUP BY can only group by columns of the source table but PostgreSQL extends this to also allow GROUP BY to group by select columns in the query select list. Grouping by value expressions instead of simple column names is also allowed.

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ...
 HAVING boolean_expression
```

If a table has been grouped using a GROUP BY clause, but then only certain groups are of interest, the HAVING clause can be used, much like a WHERE clause, to eliminate groups from a grouped table. PostgreSQL allows a HAVING clause to be used without a GROUP BY, in which case it acts like another

WHERE clause, but the point in using HAVING that way is not clear. A good rule of thumb is that a HAVING condition should refer to the results of aggregate functions. A restriction that does not involve an aggregate is more efficiently expressed in the WHERE clause.

Example:

```
SELECT pid    AS "Products",
       p.name AS "Over 5000",
       (sum(s.units) * (p.price - p.cost)) AS "Past Month Profit"
  FROM products p LEFT JOIN sales s USING ( pid )
  WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
  GROUP BY pid, p.name, p.price, p.cost
    HAVING sum(p.price * s.units) > 5000;
```

In the example above, the WHERE clause is selecting rows by a column that is not grouped, while the HAVING clause restricts the output to groups with total gross sales over 5000.

2.3. Select Lists

As shown in the previous section, the table expression in the `SELECT` command constructs an intermediate virtual table by possibly combining tables, views, eliminating rows, grouping, etc. This table is finally passed on to processing by the *select list*. The select list determines which *columns* of the intermediate table are actually output. The simplest kind of select list is `*` which emits all columns that the table expression produces. Otherwise, a select list is a comma-separated list of value expressions (as defined in Section 1.3, “Value Expressions”). For instance, it could be a list of column names:

```
SELECT a, b, c FROM ...
```

The columns names a, b, and c are either the actual names of the columns of tables referenced in the FROM clause, or the aliases given to them as explained in Section 2.2.1.3, “Table and Column Aliases”. The name space available in the select list is the same as in the WHERE clause (unless grouping is used, in which case it is the same as in the HAVING clause). If more than one table has a column of the same name, the table name must also be given, as in

```
SELECT tbl1.a, tbl2.b, tbl1.c FROM ...
```

(see also Section 2.2.2, “WHERE clause”).

If an arbitrary value expression is used in the select list, it conceptually adds a new virtual column to the returned table. The value expression is evaluated once for each retrieved row, with the row's values substituted for any column references. But the expressions in the select list do not have to reference any columns in the table expression of the FROM clause; they could be constant arithmetic expressions as well, for instance.

2.3.1. Column Labels

The entries in the select list can be assigned names for further processing. The “further processing” in this case is an optional sort specification and the client application (e.g., column headers for display). For example:

```
SELECT a AS value, b + c AS sum FROM ...
```

If no output column name is specified via AS, the system assigns a default name. For simple column references, this is the name of the referenced column. For function calls, this is the name of the function. For complex expressions, the system will generate a generic name.

Note

The naming of output columns here is different from that done in the FROM clause (see Section 2.2.1.3, “Table and Column Aliases”). This pipeline will in fact allow you to rename the same column twice, but the name chosen in the select list is the one that will be passed on.

2.3.2. DISTINCT

After the select list has been processed, the result table may optionally be subject to the elimination of duplicates. The `DISTINCT` key word is written directly after the `SELECT` to enable this:

```
SELECT DISTINCT select_list ...
```

(Instead of `DISTINCT` the word `ALL` can be used to select the default behavior of retaining all rows.)

Obviously, two rows are considered distinct if they differ in at least one column value. NULLs are considered equal in this comparison.

Alternatively, an arbitrary expression can determine what rows are to be considered distinct:

```
SELECT DISTINCT ON (expression [, expression ...]) select_list ...
```

Here *`expression`* is an arbitrary value expression that is evaluated for all rows. A set of rows for which all the expressions are equal are considered duplicates, and only the first row of the set is kept in the output. Note that the “first row” of a set is unpredictable unless the query is sorted on enough columns to guarantee a unique ordering of the rows arriving at the DISTINCT filter. (DISTINCT ON processing occurs after ORDER BY sorting.)

The DISTINCT ON clause is not part of the SQL standard and is sometimes considered bad style because of the potentially indeterminate nature of its results. With judicious use of GROUP BY and subselects in FROM the construct can be avoided, but it is very often the most convenient alternative.

2.4. Combining Queries

The results of two queries can be combined using the set operations union, intersection, and difference. The syntax is

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

`query1` and *`query2`* are queries that can use any of the features discussed up to this point. Set operations can also be nested and chained, for example

```
query1 UNION query2 UNION query3
```

which really says

```
(query1 UNION query2) UNION query3
```

`UNION` effectively appends the result of *`query2`* to the result of *`query1`* (although there is no guarantee that this is the order in which the rows are actually returned). Furthermore, it eliminates all duplicate rows, in the sense of DISTINCT, unless ALL is specified.

`INTERSECT` returns all rows that are both in the result of *`query1`* and in the result of *`query2`*. Duplicate rows are eliminated unless ALL is specified.

EXCEPT returns all rows that are in the result of *query1* but not in the result of *query2*. Again, duplicates are eliminated unless ALL is specified.

In order to calculate the union, intersection, or difference of two queries, the two queries must be "union compatible", which means that they both return the same number of columns, and that the corresponding columns have compatible data types, as described in Section 5.6, "UNION and CASE Constructs".

2.5. Sorting Rows

After a query has produced an output table (after the select list has been processed) it can optionally be sorted. If sorting is not chosen, the rows will be returned in random order. The actual order in that case will depend on the scan and join plan types and the order on disk, but it must not be relied on. A particular output ordering can only be guaranteed if the sort step is explicitly chosen.

The ORDER BY clause specifies the sort order:

```
SELECT select_list
    FROM table_expression
    ORDER BY column1 [ASC | DESC] [, column2 [ASC | DESC] ...]
```

column1, etc., refer to select list columns. These can be either the output name of a column (see Section 2.3.1, "Column Labels") or the number of a column. Some examples:

```
SELECT a, b FROM table1 ORDER BY a;
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, sum(b) FROM table1 GROUP BY a ORDER BY 1;
```

As an extension to the SQL standard, PostgreSQL also allows ordering by arbitrary expressions:

```
SELECT a, b FROM table1 ORDER BY a + b;
```

References to column names in the FROM clause that are renamed in the select list are also allowed:

```
SELECT a AS b FROM table1 ORDER BY a;
```

But these extensions do not work in queries involving UNION, INTERSECT, or EXCEPT, and are not portable to other DBMS.

Each column specification may be followed by an optional ASC or DESC to set the sort direction. ASC is default. Ascending order puts smaller values first, where "smaller" is defined in terms of the < operator. Similarly, descending order is determined with the > operator.

If more than one sort column is specified, the later entries are used to sort rows that are equal under the order imposed by the earlier sort specifications.

2.6. LIMIT and OFFSET

```
SELECT select_list
    FROM table_expression
    [LIMIT { number | ALL }] [OFFSET number]
```

LIMIT allows you to retrieve just a portion of the rows that are generated by the rest of the query. If a limit count is given, no more than that many rows will be returned. LIMIT ALL is the same as omitting a LIMIT clause.

OFFSET says to skip that many rows before beginning to return rows to the client. OFFSET 0 is the same as omitting an OFFSET clause. If both OFFSET and LIMIT appear, then OFFSET rows are skipped before starting to count the LIMIT rows that are returned.

When using LIMIT, it is a good idea to use an ORDER BY clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows---you may be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? The ordering is unknown, unless you specified ORDER BY.

The query optimizer takes LIMIT into account when generating a query plan, so you are very likely to get different plans (yielding different row orders) depending on what you give for LIMIT and OFFSET. Thus, using different LIMIT/OFFSET values to select different subsets of a query result *will give inconsistent results* unless you enforce a predictable result ordering with ORDER BY. This is not a bug; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless ORDER BY is used to constrain the order.

Chapter 3. Data Types

PostgreSQL has a rich set of native data types available to users. Users may add new types to PostgreSQL using the `CREATE TYPE` command.

Table 3.1, "Data Types" shows all general-purpose data types included in the standard distribution. Most of the alternative names listed in the "Aliases" column are the names used internally by PostgreSQL for historical reasons. In addition, some internally used or deprecated types are available, but they are not listed here.

Table 3.1. Data Types

Type Name	Aliases	Description
`bigint`	`int8`	signed eight-byte integer
`bigserial`	`serial8`	autoincrementing eight-byte integer
`bit`		fixed-length bit string
`bit varying(`*n*`)`	`varbit(`*n*`)`	variable-length bit string
`boolean`	`bool`	logical Boolean (true/false)
`box`		rectangular box in 2D plane
`bytea`		binary data
`character(`*n*`)`	`char(`*n*`)`	fixed-length character string
`character varying(`*n*`)`	`varchar(`*n*`)`	variable-length character string
`cidr`		IP network address
`circle`		circle in 2D plane
`date`		calendar date (year, month, day)
`double precision`	`float8`	double precision floating-point number
`inet`		IP host address
`integer`	`int`, `int4`	signed four-byte integer
`interval(`*p*`)`		general-use time span
`line`		infinite line in 2D plane
`lseg`		line segment in 2D plane
`macaddr`		MAC address
`money`		US-style currency
`numeric [ (`*p*`, `*s*`) ]`	`decimal [ (`*p*`, `*s*`) ]`	exact numeric with selectable precision
`oid`		object identifier
`path`		open and closed geometric path in 2D plane
`point`		geometric point in 2D plane
`polygon`		closed geometric path in 2D plane

Type Name	Aliases	Description
real	float4	single precision floating-point number
smallint	int2	signed two-byte integer
serial	serial4	autoincrementing four-byte integer
text		variable-length character string
time [(*p*)] [without time zone]		time of day
time [(*p*)] with time zone	timetz	time of day, including time zone
timestamp [(*p*)] without time zone	timestamp	date and time
timestamp [(*p*)] [with time zone]	timestamptz	date and time, including time zone

Compatibility

The following types (or spellings thereof) are specified by SQL: bit, bit varying, boolean, char, character, character varying, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time, timestamp (both with or without time zone).

Each data type has an external representation determined by its input and output functions. Many of the built-in types have obvious external formats. However, several types are either unique to PostgreSQL, such as open and closed paths, or have several possibilities for formats, such as the date and time types. Most of the input and output functions corresponding to the base types (e.g., integers and floating-point numbers) do some error-checking. Some of the input and output functions are not invertible. That is, the result of an output function may lose precision when compared to the original input.

Some of the operators and functions (e.g., addition and multiplication) do not perform run-time error-checking in the interests of improving execution speed. On some systems, for example, the numeric operators for some data types may silently underflow or overflow.

3.1. Numeric Types

Numeric types consist of two-, four-, and eight-byte integers, four- and eight-byte floating-point numbers and fixed-precision decimals.

Table 3.2. Numeric Types

Type name	Storage size	Description	Range
smallint	2 bytes	Fixed-precision	-32768 to +32767
integer	4 bytes	Usual choice for fixed-precision	-2147483648 to +2147483647
bigint	8 bytes	Very large range fixed-precision	-9223372036854775808 to 9223372036854775807

Type name	Storage size	Description	Range
`decimal`	variable	user-specified precision, exact	no limit
`numeric`	variable	user-specified precision, exact	no limit
`real`	4 bytes	variable-precision, inexact	6 decimal digits precision
`double precision`	8 bytes	variable-precision, inexact	15 decimal digits precision
`serial`	4 bytes	autoincrementing integer	1 to 2147483647
`bigserial`	8 bytes	autoincrementing integer	1 to 9223372036854775807

The syntax of constants for the numeric types is described in Section 1.1.2, “Constants”. The numeric types have a full set of corresponding arithmetic operators and functions. Refer to Chapter 4, *Functions and Operators* for more information. The following sections describe the types in detail.

3.1.1. The Integer Types

The types `smallint`, `integer`, `bigint` store whole numbers, that is, numbers without fractional components, of various ranges. Attempts to store values outside of the allowed range will result in an error.

The type `integer` is the usual choice, as it offers the best balance between range, storage size, and performance. The `smallint` type is generally only used if disk space is at a premium. The `bigint` type should only be used if the `integer` range is not sufficient, because the latter is definitely faster.

The `bigint` type may not function correctly on all platforms, since it relies on compiler support for eight-byte integers. On a machine without such support, `bigint` acts the same as `integer` (but still takes up eight bytes of storage). However, we are not aware of any reasonable platform where this is actually the case.

SQL only specifies the integer types `integer` (or `int`) and `smallint`. The type `bigint`, and the type names `int2`, `int4`, and `int8` are extensions, which are shared with various other RDBMS products.

Note

If you have a column of type `smallint` or `bigint` with an index, you may encounter problems getting the system to use that index. For instance, a clause of the form

```
... WHERE smallint_column = 42
```

will not use an index, because the system assigns type `integer` to the constant 42, and PostgreSQL currently cannot use an index when two different data types are involved. A workaround is to single-quote the constant, thus:

```
... WHERE smallint_column = '42'
```

This will cause the system to delay type resolution and will assign the right type to the constant.

3.1.2. Arbitrary Precision Numbers

The type `numeric` can store numbers of practically unlimited size and precision, while being able to store all numbers and carry out all calculations exactly. It is especially recommended for storing monetary

amounts and other quantities where exactness is required. However, the `numeric` type is very slow compared to the floating-point types described in the next section.

In what follows we use these terms: The *scale* of a `numeric` is the count of decimal digits in the fractional part, to the right of the decimal point. The *precision* of a `numeric` is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

Both the precision and the scale of the numeric type can be configured. To declare a column of type `numeric` use the syntax

```
NUMERIC(precision, scale)
```

The precision must be positive, the scale zero or positive. Alternatively,

```
NUMERIC(precision)
```

selects a scale of 0. Specifying

```
NUMERIC
```

without any precision or scale creates a column in which numeric values of any precision and scale can be stored, up to the implementation limit on precision. A column of this kind will not coerce input values to any particular scale, whereas `numeric` columns with a declared scale will coerce input values to that scale. (The SQL standard requires a default scale of 0, i.e., coercion to integer accuracy. We find this a bit useless. If you're concerned about portability, always specify the precision and scale explicitly.)

If the precision or scale of a value is greater than the declared precision or scale of a column, the system will attempt to round the value. If the value cannot be rounded so as to satisfy the declared limits, an error is raised.

The types `decimal` and `numeric` are equivalent. Both types are part of the SQL standard.

3.1.3. Floating-Point Types

The data types `real` and `double precision` are inexact, variable-precision numeric types. In practice, these types are usually implementations of IEEE 754 binary floating point (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and printing back out a value may show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and will not be discussed further here, except for the following points:

- If you require exact storage and calculations (such as for monetary amounts), use the `numeric` type instead.
- If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.
- Comparing two floating-point values for equality may or may not work as expected.

Normally, the `real` type has a range of at least -1E+37 to +1E+37 with a precision of at least 6 decimal digits. The `double precision` type normally has a range of around -1E+308 to +1E+308 with a

precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding may take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

3.1.4. The Serial Types

The `serial` data types are not truly types, but are a notational convenience for setting up unique identifier columns in tables. In the current implementation, specifying

```
CREATE TABLE tablename (
    colname SERIAL
);
```

is equivalent to specifying:

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename (
    colname integer DEFAULT nextval('tablename_colname_seq') UNIQUE
 NOT NULL
);
```

Thus, we have created an integer column and arranged for its default values to be assigned from a sequence generator. UNIQUE and NOT NULL constraints are applied to ensure that explicitly-inserted values will never be duplicates, either.

The type names `serial` and `serial4` are equivalent: both create `integer` columns. The type names `bigserial` and `serial8` work just the same way, except that they create a `bigint` column. `bigserial` should be used if you anticipate use of more than 2^{31} identifiers over the lifetime of the table.

Implicit sequences supporting the `serial` types are not automatically dropped when a table containing a serial type is dropped. So, the following commands executed in order will likely fail:

```
CREATE TABLE tablename (colname SERIAL);
DROP TABLE tablename;
CREATE TABLE tablename (colname SERIAL);
```

The sequence will remain in the database until explicitly dropped using `DROP SEQUENCE`. (This annoyance will probably be changed in some future release.)

3.2. Monetary Type

Deprecated

The `money` type is deprecated. Use `numeric` or `decimal` instead, in combination with the `to_char` function. The money type may become a locale-aware layer over the `numeric` type in a future release.

The `money` type stores U.S.-style currency with fixed decimal point representation. If PostgreSQL is compiled with locale support then the `money` type uses locale-specific output formatting.

Input is accepted in a variety of formats, including integer and floating-point literals, as well as “typical” currency formatting, such as `'$1,000.00'`. Output is in the latter form.

Table 3.3. Monetary Types

Type Name	Storage	Description	Range
money	4 bytes	Fixed-precision	-21474836.48 to +21474836.47

3.3. Character Types

Table 3.4. Character Types

Type name	Description
`character(n)`, `char(n)`	Fixed-length blank padded
`character varying(n)`, `varchar(n)`	Variable-length with limit
`text`	Variable unlimited length

SQL defines two primary character types: `character(n)` and `character varying(n)`, where *n* is a positive integer. Both of these types can store strings up to *n* characters in length. An attempt to store a longer string into a column of these types will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length. (This somewhat bizarre exception is required by the SQL standard.) If the string to be stored is shorter than the declared length, values of type `character` will be space-padded; values of type `character varying` will simply store the shorter string.

Note

Prior to PostgreSQL 7.2, strings that were too long were silently truncated, no error was raised.

The notations `char(n)` and `varchar(n)` are aliases for `character(n)` and `character varying(n)`, respectively. `character` without length specifier is equivalent to `character(1)`; if `character varying` is used without length specifier, the type accepts strings of any size. The latter is a PostgreSQL extension.

In addition, PostgreSQL supports the more general `text` type, which stores strings of any length. Unlike `character varying`, `text` does not require an explicit declared upper limit on the size of the string. Although the type `text` is not in the SQL standard, many other RDBMS packages have it as well.

The storage requirement for data of these types is 4 bytes plus the actual string, and in case of `character` plus the padding. Long strings will be compressed by the system automatically, so the physical requirement on disk may be less. In any case, the longest possible character string that can be stored is about 1 GB. (The maximum value that will be allowed for *n* in the data type declaration is less than that. It wouldn't be very useful to change this because with multibyte character encodings the number of characters and bytes can be quite different anyway. If you desire to store long strings with no specific upper limit, use `text` or `character varying` without a length specifier, rather than making up an arbitrary length limit.)

Tip

There are no performance differences between these three types, apart from the increased storage size when using the blank-padded type.

Refer to Section 1.1.2.1, “String Constants” for information about the syntax of string literals, and to Chapter 4, *Functions and Operators* for information about available operators and functions.

Example 3.1. Using the character types

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- 1
  a   | char_length
------+-------------
 ok   |           4

CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good      ');
INSERT INTO test2 VALUES ('too long');
ERROR:  value too long for type character varying(5)
SELECT b, char_length(b) FROM test2;
   b   | char_length
-------+-------------
 ok    |           2
 good  |           5
```

1 The `char_length` function is discussed in Section 4.4, “String Functions and Operators”.

There are two other fixed-length character types in PostgreSQL. The `name` type exists *only* for storage of internal catalog names and is not intended for use by the general user. Its length is currently defined as 32 bytes (31 usable characters plus terminator) but should be referenced using the macro `NAMEDATALEN`. The length is set at compile time (and is therefore adjustable for special uses); the default maximum length may change in a future release. The type `"char"` (note the quotes) is different from `char(1)` in that it only uses one byte of storage. It is internally used in the system catalogs as a poor-man's enumeration type.

Table 3.5. Specialty Character Type

Type Name	Storage	Description
"char"	1 byte	Single character internal type
name	32 bytes	Thirty-one character internal type

3.4. Binary Strings

The `bytea` data type allows storage of binary strings.

Table 3.6. Binary String Types

Type Name	Storage	Description
`bytea`	4 bytes plus the actual binary string	Variable (not specifically limited) length binary string

A binary string is a sequence of octets that does not have either a character set or collation associated with it. `Bytea` specifically allows storing octets of zero value and other “non-printable” octets.

Octets of certain values *must* be escaped (but all octet values *may* be escaped) when used as part of a string literal in an SQL statement. In general, to escape an octet, it is converted into the three-digit octal number equivalent of its decimal octet value, and preceded by two backslashes. Some octet values have alternate escape sequences, as shown in Table 3.7, “SQL Literal Escaped Octets”.

Table 3.7. SQL Literal Escaped Octets

Decimal Octet Value	Description	Input Escaped Representation	Example	Printed Result
0	zero octet	'\\000'	select '\\000'::bytea;	\000
39	single quote	'\'' or '\047'	select '\''::bytea;	'
92	backslash	'\\\\' or '\134'	select '\\\\'::bytea;	\\

Note that the result in each of the examples above was exactly one octet in length, even though the output representation of the zero octet and backslash are more than one character. `Bytea` output octets are also escaped. In general, each "non-printable" octet decimal value is converted into its equivalent three digit octal value, and preceded by one backslash. Most "printable" octets are represented by their standard representation in the client character set. The octet with decimal value 92 (backslash) has a special alternate output representation. Details are in Table 3.8, "SQL Output Escaped Octets".

Table 3.8. SQL Output Escaped Octets

Decimal Octet Value	Description	Output Escaped Representation	Example	Printed Result
92	backslash	\\	select '\\134'::bytea;	\\
0 to 31 and 127 to 255	"non-printable" octets	\### (octal value)	select '\001'::bytea;	\001
32 to 126	"printable" octets	ASCII representation	select '\176'::bytea;	~

SQL string literals (input strings) must be preceded with two backslashes due to the fact that they must pass through two parsers in the PostgreSQL backend. The first backslash is interpreted as an escape character by the string-literal parser, and therefore is consumed, leaving the octets that follow. The remaining backslash is recognized by the `bytea` input function as the prefix of a three digit octal value. For example, a string literal passed to the backend as `'\\001'` becomes `'\001'` after passing through the string-literal parser. The `'\001'` is then sent to the `bytea` input function, where it is converted to a single octet with a decimal value of 1.

For a similar reason, a backslash must be input as `'\\\\'` (or `'\\134'`). The first and third backslashes are interpreted as escape characters by the string-literal parser, and therefore are consumed, leaving two backslashes in the string passed to the `bytea` input function, which interprets them as representing a single backslash. For example, a string literal passed to the backend as `'\\\\'` becomes `'\\'` after passing through the string-literal parser. The `'\\'` is then sent to the `bytea` input function, where it is converted to a single octet with a decimal value of 92.

A single quote is a bit different in that it must be input as `'\''` (or `'\\134'`), *not* as `'\\''`. This is because, while the literal parser interprets the single quote as a special character, and will consume the single backslash, the `bytea` input function does *not* recognize a single quote as a special octet. Therefore a string literal passed to the backend as `'\''` becomes `''''` after passing through the string-literal parser. The `''''` is then sent to the `bytea` input function, where it is retains its single octet decimal value of 39.

Depending on the front end to PostgreSQL you use, you may have additional work to do in terms of escaping and unescaping `bytea` strings. For example, you may also have to escape line feeds and carriage

returns if your interface automatically translates these. Or you may have to double up on backslashes if the parser for your language or choice also treats them as an escape character.

`Bytea` provides most of the functionality of the binary string type per SQL99 section 4.3. A comparison of SQL99 Binary Strings and PostgreSQL `bytea` is presented in Table 3.9, "Comparison of SQL99 Binary String and PostgreSQL `BYTEA` types".

Table 3.9. Comparison of SQL99 Binary String and PostgreSQL `BYTEA` types

SQL99	`BYTEA`
Name of data type `BINARY LARGE OBJECT` or `BLOB`	Name of data type `BYTEA`
Sequence of octets that does not have either a character set or collation associated with it.	same
Described by a binary data type descriptor containing the name of the data type and the maximum length in octets	Described by a binary data type descriptor containing the name of the data type with no specific maximum length
All binary strings are mutually comparable in accordance with the rules of comparison predicates.	same
Binary string values can only be compared for equality.	Binary string values can be compared for equality, greater than, greater than or equal, less than, less than or equal
Operators operating on and returning binary strings include concatenation, substring, overlay, and trim	Operators operating on and returning binary strings include concatenation, substring, and trim. The `leading` and `trailing` arguments for trim are not yet implemented.
Other operators involving binary strings include length, position, and the like predicate	same
A binary string literal is comprised of an even number of hexadecimal digits, in single quotes, preceded by "X", e.g. `X'1a43fe'`	A binary string literal is comprised of octets escaped according to the rules shown in Table 3.7, "SQL Literal Escaped Octets"

3.5. Date/Time Types

PostgreSQL supports the full set of SQL date and time types.

Table 3.10. Date/Time Types

Type	Description	Storage	Earliest	Latest	Resolution
`timestamp [ (p) ] without time zone`	both date and time	8 bytes	4713 BC	AD 1465001	1 microsecond / 14 digits
`timestamp [ (p) ] [ with time zone ]`	both date and time	8 bytes	4713 BC	AD 1465001	1 microsecond / 14 digits
`interval [ (p) ]`	for time intervals	12 bytes	-178000000 years	178000000 years	1 microsecond

Type	Description	Storage	Earliest	Latest	Resolution
date	dates only	4 bytes	4713 BC	32767 AD	1 day
time [(p)] [without time zone]	times of day only	8 bytes	00:00:00.00	23:59:59.99	1 microsecond
time [(p)] with time zone	times of day only	12 bytes	00:00:00.00+12	23:59:59.99-12	1 microsecond

time, timestamp, and interval accept an optional precision value *p* which specifies the number of fractional digits retained in the seconds field. By default, there is no explicit bound on precision. The effective limit of precision is determined by the underlying double precision floating-point number used to store values (in seconds for interval and in seconds since 2000-01-01 for timestamp). The useful range of *p* is from 0 to about 6 for timestamp, but may be more for interval. The system will accept *p* ranging from 0 to 13.

Time zones, and time-zone conventions, are influenced by political decisions, not just earth geometry. Time zones around the world became somewhat standardized during the 1900's, but continue to be prone to arbitrary changes. PostgreSQL uses your operating system's underlying features to provide output time-zone support, and these systems usually contain information for only the time period 1902 through 2038 (corresponding to the full range of conventional Unix system time). timestamp with time zone and time with time zone will use time zone information only within that year range, and assume that times outside that range are in UTC.

To ensure an upgrade path from versions of PostgreSQL earlier than 7.0, we recognize datetime (equivalent to timestamp) and timespan (equivalent to interval). These types are now restricted to having an implicit translation to timestamp and interval, and support for these will be removed in the next release of PostgreSQL (likely named 7.3).

The types abstime and reltime are lower precision types which are used internally. You are discouraged from using any of these types in new applications and are encouraged to move any old ones over when appropriate. Any or all of these internal types might disappear in a future release.

3.5.1. Date/Time Input

Date and time input is accepted in almost any reasonable format, including ISO 8601, SQL-compatible, traditional PostgreSQL, and others. For some formats, ordering of month and day in date input can be ambiguous and there is support for specifying the expected ordering of these fields. The command SET DateStyle TO 'US' or SET DateStyle TO 'NonEuropean' specifies the variant “month before day”, the command SET DateStyle TO 'European' sets the variant “day before month”. The ISO style is the default but this default can be changed at compile time or at run time.

PostgreSQL is more flexible in handling date/time than the SQL standard requires. See Appendix A, *Date/Time Support* for the exact parsing rules of date/time input and for the recognized text fields including months, days of the week, and time zones.

Remember that any date or time literal input needs to be enclosed in single quotes, like text strings. Refer to Section 1.1.2.5, “Constants of Other Types” for more information. SQL9x requires the following syntax

type [(*p*)] '*value*'

where *p* in the optional precision specification is an integer corresponding to the number of fractional digits in the seconds field. Precision can be specified for `time`, `timestamp`, and `interval` types.

3.5.1.1. `date`

The following are some possible inputs for the `date` type.

Table 3.11. Date Input

Example	Description
January 8, 1999	Unambiguous
1999-01-08	ISO-8601 format, preferred
1/8/1999	U.S.; read as August 1 in European mode
8/1/1999	European; read as August 1 in U.S. mode
1/18/1999	U.S.; read as January 18 in any mode
19990108	ISO-8601 year, month, day
990108	ISO-8601 year, month, day
1999.008	Year and day of year
99008	Year and day of year
J2451187	Julian day
January 8, 99 BC	Year 99 before the Common Era

3.5.1.2. `time [ ( p ) ] [ without time zone ]`

Per SQL99, this type can be specified as `time` or as `time without time zone`. The optional precision *p* should be between 0 and 13, and defaults to the precision of the input time literal.

The following are valid `time` inputs.

Table 3.12. Time Input

Example	Description
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	Same as 04:05; AM does not affect value
04:05 PM	Same as 16:05; input hour must be <= 12
allballs	Same as 00:00:00

3.5.1.3. `time [ ( precision ) ] with time zone`

This type is defined by SQL92, but the definition exhibits properties which lead to questionable usefulness. In most cases, a combination of `date`, `time`, `timestamp without time zone` and `timestamp with time zone` should provide a complete range of date/time functionality required by any application.

The optional precision *p* should be between 0 and 13, and defaults to the precision of the input time literal.

`time with time zone` accepts all input also legal for the `time` type, appended with a legal time zone, as follows:

Table 3.13. Time With Time Zone Input

Example	Description
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601

Refer to Table 3.14, “Time Zone Input” for more examples of time zones.

3.5.1.4. `timestamp [ (`*precision*`) ] without time zone`

Valid input for the `timestamp [ (`*p*`) ] without time zone` type consists of a concatenation of a date and a time, followed by an optional `AD` or `BC`, followed by an optional time zone. (See below.) Thus

```
1999-01-08 04:05:06
```

is a valid `timestamp without time zone` value that is ISO-compliant. In addition, the wide-spread format

```
January 8 04:05:06 1999 PST
```

is supported.

The optional precision *p* should be between 0 and 13, and defaults to the precision of the input `timestamp` literal.

For `timestamp without time zone`, any explicit time zone specified in the input is silently swallowed. That is, the resulting date/time value is derived from the explicit date/time fields in the input value, and is not adjusted for time zone.

3.5.1.5. `timestamp [ (`*precision*`) ] with time zone`

Valid input for the `timestamp` type consists of a concatenation of a date and a time, followed by an optional `AD` or `BC`, followed by an optional time zone. (See below.) Thus

```
1999-01-08 04:05:06 -8:00
```

is a valid `timestamp` value that is ISO-compliant. In addition, the wide-spread format

```
January 8 04:05:06 1999 PST
```

is supported.

The optional precision *p* should be between 0 and 13, and defaults to the precision of the input `timestamp` literal.

Table 3.14. Time Zone Input

Time Zone	Description
PST	Pacific Standard Time

Time Zone	Description
-8:00	ISO-8601 offset for PST
-800	ISO-8601 offset for PST
-8	ISO-8601 offset for PST

3.5.1.6. `interval [ ( precision ) ]`

`interval` values can be written with the following syntax:

```
  Quantity Unit [Quantity Unit...] [Direction]
@ Quantity Unit [Quantity Unit...] [Direction]
```

where: `Quantity` is a number (possibly signed), `Unit` is `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, `millennium`, or abbreviations or plurals of these units; `Direction` can be `ago` or empty. The at sign (`@`) is optional noise. The amounts of different units are implicitly added up with appropriate sign accounting.

Quantities of days, hours, minutes, and seconds can be specified without explicit unit markings. For example, `'1 12:59:10'` is read the same as `'1 day 12 hours 59 min 10 sec'`.

The optional precision *p* should be between 0 and 13, and defaults to the precision of the input literal.

3.5.1.7. Special values

The following SQL-compatible functions can be used as date or time input for the corresponding data type: `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`. The latter two accept an optional precision specification.

PostgreSQL also supports several special constants for convenience.

Table 3.15. Special Date/Time Constants

Constant	Description
epoch	1970-01-01 00:00:00+00 (Unix system time zero)
infinity	Later than other valid times
-infinity	Earlier than other valid times
invalid	Illegal entry
now	Current transaction time
today	Midnight today
tomorrow	Midnight tomorrow
yesterday	Midnight yesterday
zulu, allballs, z	00:00:00.00 GMT

`'now'` is evaluated when the value is first interpreted.

Note

As of PostgreSQL version 7.2, `'current'` is no longer supported as a date/time constant. Previously, `'current'` was stored as a special value, and evaluated to `'now'` only when used in an expression or type conversion.

3.5.2. Date/Time Output

Output formats can be set to one of the four styles ISO 8601, SQL (Ingres), traditional PostgreSQL, and German, using the `SET DateStyle`. The default is the ISO format.

Table 3.16. Date/Time Output Styles

Style Specification	Description	Example
'ISO'	ISO-8601 standard	1997-12-17 07:37:16-08
'SQL'	Traditional style	12/17/1997 07:37:16.00 PST
'PostgreSQL'	Original style	Wed Dec 17 07:37:16 1997 PST
'German'	Regional style	17.12.1997 07:37:16.00 PST

The output of the `date` and `time` styles is of course only the date or time part in accordance with the above examples.

The SQL style has European and non-European (U.S.) variants, which determines whether month follows day or vice versa. (See also Section 3.5.1, “Date/Time Input” for how this setting affects interpretation of input values.)

Table 3.17. Date-Order Conventions

Style Specification	Description	Example
European	*day*/*month*/*year*	17/12/1997 15:37:16.00 MET
US	*month*/*day*/*year*	12/17/1997 07:37:16.00 PST

`interval` output looks like the input format, except that units like `week` or `century` are converted to years and days. In ISO mode the output looks like

```
[ Quantity Units [ ... ] ] [ Days ] Hours:Minutes [ ago ]
```

There are several ways to affect the appearance of date/time types:

- The `PGDATESTYLE` environment variable used by the backend directly on postmaster start-up.
- The `PGDATESTYLE` environment variable used by the frontend libpq on session start-up.
- `SET DATESTYLE` SQL command.

3.5.3. Time Zones

PostgreSQL endeavors to be compatible with SQL92 definitions for typical usage. However, the SQL92 standard has an odd mix of date and time types and capabilities. Two obvious problems are:

- Although the `date` type does not have an associated time zone, the `time` type can. Time zones in the real world can have no meaning unless associated with a date as well as a time since the offset may vary through the year with daylight-saving time boundaries.

- The default time zone is specified as a constant integer offset from GMT/UTC. It is not possible to adapt to daylight-saving time when doing date/time arithmetic across DST boundaries.

To address these difficulties, we recommend using date/time types that contain both date and time when using time zones. We recommend *not* using the SQL92 type `time with time zone`

(though it is supported by PostgreSQL for legacy applications and for compatibility with other RDBMS implementations). PostgreSQL assumes your local time zone for any type containing only date or time. Further, time zone support is derived from the underlying operating system time-zone capabilities, and hence can handle daylight-saving time and other expected behavior.

PostgreSQL obtains time-zone support from the underlying operating system for dates between 1902 and 2038 (near the typical date limits for Unix-style systems). Outside of this range, all dates are assumed to be specified and used in Universal Coordinated Time (UTC).

All dates and times are stored internally in UTC, traditionally known as Greenwich Mean Time (GMT). Times are converted to local time on the database server before being sent to the client frontend, hence by default are in the server time zone.

There are several ways to affect the time-zone behavior:

- The `TZ` environment variable is used by the backend directly on postmaster start-up as the default time zone.
- The `PGTZ` environment variable, if set at the client, is used by libpq to send a `SET TIME ZONE` command to the backend upon connection.
- The SQL command `SET TIME ZONE` sets the time zone for the session.
- The SQL92 qualifier on

  `timestamp` `AT TIME ZONE '`*`zone`*`'`

  where *`zone`* can be specified as a text time zone (e.g. `'PST'`) or as an interval (e.g. `INTERVAL '-08:00'`).

Note

If an invalid time zone is specified, the time zone becomes GMT (on most systems anyway).

Note

If the runtime option `AUSTRALIAN_TIMEZONES` is set then `CST` and `EST` refer to Australian time zones, not American ones.

3.5.4. Internals

PostgreSQL uses Julian dates for all date/time calculations. They have the nice property of correctly predicting/calculating any date more recent than 4713BC to far into the future, using the assumption that the length of the year is 365.2425 days.

Date conventions before the 19th century make for interesting reading, but are not consistent enough to warrant coding into a date/time handler.

3.6. Boolean Type

PostgreSQL provides the SQL99 type `boolean`. `boolean` can have one of only two states: “true” or “false”. A third state, “unknown”, is represented by the SQL NULL state.

Valid literal values for the “true” state are:

```
TRUE
't'
```

```
'true'
'y'
'yes'
'1'
```

For the “false” state, the following values can be used:

```
FALSE
'f'
'false'
'n'
'no'
'0'
```

Using the key words `TRUE` and `FALSE` is preferred (and SQL-compliant).

Example 3.2. Using the `boolean` type

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

Example 3.2, “Using the `boolean` type” shows that `boolean` values are output using the letters `t` and `f`.

Tip

Values of the `boolean` type cannot be cast directly to other types (e.g., `CAST (`*`boolval`* `AS integer)` does not work). This can be accomplished using the `CASE` expression: `CASE WHEN` *`boolval`* `THEN 'value if true' ELSE 'value if false' END`. See also Section 4.12, “Conditional Expressions”.

`boolean` uses 1 byte of storage.

3.7. Geometric Types

Geometric types represent two-dimensional spatial objects. The most fundamental type, the point, forms the basis for all of the other types.

Table 3.18. Geometric Types

Geometric Type	Storage	Representation	Description
`point`	16 bytes	(x,y)	Point in space
`line`	32 bytes	((x1,y1),(x2,y2))	Infinite line
`lseg`	32 bytes	((x1,y1),(x2,y2))	Finite line segment

Geometric Type	Storage	Representation	Description
`box`	32 bytes	((x1,y1),(x2,y2))	Rectangular box
`path`	4+32n bytes	((x1,y1),...)	Closed path (similar to polygon)
`path`	4+32n bytes	[(x1,y1),...]	Open path
`polygon`	4+32n bytes	((x1,y1),...)	Polygon (similar to closed path)
`circle`	24 bytes	<(x,y),r>	Circle (center and radius)

A rich set of functions and operators is available to perform various geometric operations such as scaling, translation, rotation, and determining intersections.

3.7.1. Point

Points are the fundamental two-dimensional building block for geometric types.

`point` is specified using the following syntax:

```
( x , y )
  x , y
```

where the arguments are

x

The x-axis coordinate as a floating-point number

y

The y-axis coordinate as a floating-point number

3.7.2. Line Segment

Line segments (`lseg`) are represented by pairs of points.

`lseg` is specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where the arguments are

(*x1*,*y1*)
(*x2*,*y2*)

The end points of the line segment

3.7.3. Box

Boxes are represented by pairs of points that are opposite corners of the box.

`box` is specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where the arguments are

(*x1*,*y1*)
(*x2*,*y2*)

Opposite corners of the box

Boxes are output using the first syntax. The corners are reordered on input to store the upper right corner, then the lower left corner. Other corners of the box can be entered, but the lower left and upper right corners are determined from the input and stored.

3.7.4. Path

Paths are represented by connected sets of points. Paths can be *open*, where the first and last points in the set are not connected, and *closed*, where the first and last point are connected. Functions `popen(p)` and `pclose(p)` are supplied to force a path to be open or closed, and functions `isopen(p)` and `isclosed(p)` are supplied to test for either type in a query.

`path` is specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

where the arguments are

(*x*,*y*)

End points of the line segments comprising the path. A leading square bracket ("[") indicates an open path, while a leading parenthesis ("(") indicates a closed path.

Paths are output using the first syntax.

3.7.5. Polygon

Polygons are represented by sets of points. Polygons should probably be considered equivalent to closed paths, but are stored differently and have their own set of support routines.

`polygon` is specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

where the arguments are

(*x*,*y*)

End points of the line segments comprising the boundary of the polygon

Polygons are output using the first syntax.

3.7.6. Circle

Circles are represented by a center point and a radius.

`circle` is specified using the following syntax:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

where the arguments are

(*x*,*y*)

Center of the circle

r

Radius of the circle

Circles are output using the first syntax.

3.8. Network Address Data Types

PostgreSQL offers data types to store IP and MAC addresses. It is preferable to use these types over plain text types, because these types offer input error checking and several specialized operators and functions.

Table 3.19. Network Address Data Types

Name	Storage	Description	Range
`cidr`	12 bytes	IP networks	valid IPv4 networks
`inet`	12 bytes	IP hosts and networks	valid IPv4 hosts or networks
`macaddr`	6 bytes	MAC addresses	customary formats

IP v6 is not supported, yet.

3.8.1. `inet`

The `inet` type holds an IP host address, and optionally the identity of the subnet it is in, all in one field. The subnet identity is represented by the number of bits in the network part of the address (the "netmask"). If the netmask is 32, then the value does not indicate a subnet, only a single host. Note that if you want to accept networks only, you should use the `cidr` type rather than `inet`.

The input format for this type is *x.x.x.x/y* where *x.x.x.x* is an IP address and *y* is the number of bits in the netmask. If the */y* part is left off, then the netmask is 32, and the value represents just a single host. On display, the */y* portion is suppressed if the netmask is 32.

3.8.2. cidr

The cidr type holds an IP network specification. Input and output formats follow Classless Internet Domain Routing conventions. The format for specifying classless networks is *x.x.x.x/y* where *x.x.x.x* is the network and *y* is the number of bits in the netmask. If *y* is omitted, it is calculated using assumptions from the older classful numbering system, except that it will be at least large enough to include all of the octets written in the input.

Here are some examples:

Table 3.20. cidr Type Input Examples

CIDR Input	CIDR Displayed	abbrev(CIDR)
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

3.8.3. inet vs cidr

The essential difference between inet and cidr data types is that inet accepts values with nonzero bits to the right of the netmask, whereas cidr does not.

Tip

If you do not like the output format for inet or cidr values, try the host(), text(), and abbrev() functions.

3.8.4. macaddr

The macaddr type stores MAC addresses, i.e., Ethernet card hardware addresses (although MAC addresses are used for other purposes as well). Input is accepted in various customary formats, including

```
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'08-00-2b-01-02-03'
'08:00:2b:01:02:03'
```

which would all specify the same address. Upper and lower case is accepted for the digits `a` through `f`. Output is always in the last of the shown forms.

The directory `contrib/mac` in the PostgreSQL source distribution contains tools that can be used to map MAC addresses to hardware manufacturer names.

3.9. Bit String Types

Bit strings are strings of 1's and 0's. They can be used to store or visualize bit masks. There are two SQL bit types: `BIT(`*x*`)` and `BIT VARYING(`*x*`)`; where *x* is a positive integer.

`BIT` type data must match the length *x* exactly; it is an error to attempt to store shorter or longer bit strings. `BIT VARYING` is of variable length up to the maximum length *x*; longer strings will be rejected. `BIT` without length is equivalent to `BIT(1)`, `BIT VARYING` without length specification means unlimited length.

Note

Prior to PostgreSQL 7.2, `BIT` type data was zero-padded on the right. This was changed to comply with the SQL standard. To implement zero-padded bit strings, a combination of the concatenation operator and the `substring` function can be used.

Refer to Section 1.1.2.2, “Bit-String Constants” for information about the syntax of bit string constants. Bit-logical operators and string manipulation functions are available; see Chapter 4, *Functions and Operators*.

Example 3.3. Using the bit string types

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');
ERROR:  bit string length does not match type bit(3)
SELECT SUBSTRING(b FROM 1 FOR 2) FROM test;
```

Chapter 4. Functions and Operators

PostgreSQL provides a large number of functions and operators for the built-in data types. Users can also define their own functions and operators, as described in the *Programmer's Guide*. The psql commands `\df` and `\do` can be used to show the list of all actually available functions and operators, respectively.

If you are concerned about portability then take note that most of the functions and operators described in this chapter, with the exception of the most trivial arithmetic and comparison operators and some explicitly marked functions, are not specified by the SQL standard. Some of this extended functionality is present in other RDBMS products, and in many cases this functionality is compatible and consistent between various products.

4.1. Logical Operators

The usual logical operators are available:

AND
OR
NOT
SQL uses a three-valued Boolean logic where NULL represents “unknown”. Observe the following truth tables:

a	*b*	*a* AND *b*	*a* OR *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

a	NOT *a*
TRUE	FALSE
FALSE	TRUE
NULL	NULL

4.2. Comparison Operators

Table 4.1. Comparison Operators

Operator	Description
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal

Operator	Description
<> or !=	not equal

Note

The != operator is converted to <> in the parser stage. It is not possible to implement != and <> operators that do different things.

Comparison operators are available for all data types where this makes sense. All comparison operators are binary operators that return values of type `boolean`; expressions like `1 < 2 < 3` are not valid (because there is no < operator to compare a Boolean value with `3`).

In addition to the comparison operators, the special `BETWEEN` construct is available.

a `BETWEEN` *x* `AND` *y*

is equivalent to

a `>=` *x* `AND` *a* `<=` *y*

Similarly,

a `NOT BETWEEN` *x* `AND` *y*

is equivalent to

a `<` *x* `OR` *a* `>` *y*

There is no difference between the two respective forms apart from the CPU cycles required to rewrite the first one into the second one internally.

To check whether a value is or is not NULL, use the constructs

expression `IS NULL`
expression `IS NOT NULL`

or the equivalent, but less standard, constructs

expression `ISNULL`
expression `NOTNULL`

Do *not* write *expression* `= NULL` because NULL is not “equal to” NULL. (NULL represents an unknown value, and it is not known whether two unknown values are equal.)

Some applications may (incorrectly) require that *expression* `= NULL` returns true if *expression* evaluates to the NULL value. To support these applications, the run-time option `transform_null_equals` can be turned on (e.g., `SET transform_null_equals TO ON;`). PostgreSQL will then convert `x = NULL` clauses to `x IS NULL`. This was the default behavior in releases 6.5 through 7.1.

Boolean values can also be tested using the constructs

expression `IS TRUE`
expression `IS NOT TRUE`
expression `IS FALSE`
expression `IS NOT FALSE`

```
expression IS UNKNOWN
expression IS NOT UNKNOWN
```

These are similar to `IS NULL` in that they will always return TRUE or FALSE, never NULL, even when the operand is NULL. A NULL input is treated as the logical value UNKNOWN.

4.3. Mathematical Functions and Operators

Mathematical operators are provided for many PostgreSQL types. For types without common mathematical conventions for all possible permutations (e.g. date/time types) we describe the actual behavior in subsequent sections.

Table 4.2. Mathematical Operators

Name	Description	Example	Result
+	Addition	2 + 3	5
−	Subtraction	2 - 3	-1
*	Multiplication	2 * 3	6
/	Division (integer division truncates results)	4 / 2	2
%	Modulo (remainder)	5 % 4	1
^	Exponentiation	2.0 ^ 3.0	8
| /	Square root	|/ 25.0	5
| | /	Cube root	||/ 27.0	3
!	Factorial	5 !	120
! !	Factorial (prefix operator)	!! 5	120
@	Absolute value	@ -5.0	5
&	Binary AND	91 & 15	11
|	Binary OR	32 | 3	35
#	Binary XOR	17 # 5	20
~	Binary NOT	~1	-2
<<	Binary shift left	1 << 4	16
>>	Binary shift right	8 >> 2	2

The “binary” operators are also available for the bit string types `BIT` and `BIT VARYING`.

Table 4.3. Bit String Binary Operators

Example	Result
B'10001' & B'01101'	00001
B'10001' | B'01101'	11101
B'10001' # B'01101'	11110
~ B'10001'	01110

Example	Result
B'10001' << 3	01000
B'10001' >> 2	00100

Bit string arguments to `&`, `|`, and `#` must be of equal length. When bit shifting, the original length of the string is preserved, as shown here.

Table 4.4. Mathematical Functions

Function	Return Type	Description	Example	Result
`abs(`*x*`)`	(same as x)	absolute value	`abs(-17.4)`	17.4
`cbrt(dp)`	`dp`	cube root	`cbrt(27.0)`	3
`ceil(numeric)`	`numeric`	smallest integer not less than argument	`ceil(-42.8)`	-42
`degrees(dp)`	`dp`	radians to degrees	`degrees(0.5)`	28.6478897565412
`exp(dp)`	`dp`	exponential	`exp(1.0)`	2.71828182845905
`floor(numeric)`	`numeric`	largest integer not greater than argument	`floor(-42.8)`	-43
`ln(dp)`	`dp`	natural logarithm	`ln(2.0)`	0.693147180559945
`log(dp)`	`dp`	base 10 logarithm	`log(100.0)`	2
`log(`*b* `numeric, `*x* `numeric)`	`numeric`	logarithm to base *b*	`log(2.0, 64.0)`	6.0000000000
`mod(`*y*, *x*`)`	(same as argument types)	remainder of *y*/*x*	`mod(9,4)`	1
`pi()`	`dp`	“Pi” constant	`pi()`	3.14159265358979
`pow(`*e* `dp, `*n* `dp)`	`dp`	raise a number to exponent *e*	`pow(9.0, 3.0)`	729
`radians(dp)`	`dp`	degrees to radians	`radians(45.0)`	0.785398163397448
`random()`	`dp`	value between 0.0 to 1.0	`random()`	
`round(dp)`	`dp`	round to nearest integer	`round(42.4)`	42
`round(`*v* `numeric, `*s* `integer)`	`numeric`	round to *s* decimal places	`round(42.4382, 2)`	42.44
`sign(numeric)`	`numeric`	sign of the argument (-1, 0, +1)	`sign(-8.4)`	-1
`sqrt(dp)`	`dp`	square root	`sqrt(2.0)`	1.4142135623731
`trunc(dp)`	`dp`	truncate toward zero	`trunc(42.8)`	42
`trunc(numeric, `*s* `integer)`	`numeric`	truncate to *s* decimal places	`trunc(42.4382, 2)`	42.43

In the table above, `dp` indicates `double precision`. The functions `exp`, `ln`, `log`, `pow`, `round` (1 argument), `sqrt`, and `trunc` (1 argument) are also available for the type `numeric` in place of `double`

precision. Functions returning a numeric result take numeric input arguments, unless otherwise specified. Many of these functions are implemented on top of the host system's C library; accuracy and behavior in boundary cases could therefore vary depending on the host system.

Table 4.5. Trigonometric Functions

Function	Description
acos(*x*)	inverse cosine
asin(*x*)	inverse sine
atan(*x*)	inverse tangent
atan2(*x*, *y*)	inverse tangent of *y*/*x*
cos(*x*)	cosine
cot(*x*)	cotangent
sin(*x*)	sine
tan(*x*)	tangent

All trigonometric functions have arguments and return values of type double precision.

4.4. String Functions and Operators

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of all the types CHARACTER, CHARACTER VARYING, and TEXT. Unless otherwise noted, all of the functions listed below work on all of these types, but be wary of potential effects of the automatic padding when using the CHARACTER type. Generally, the functions described here also work on data of non-string types by converting that data to a string representation first. Some functions also exist natively for bit-string types.

SQL defines some string functions with a special syntax where certain keywords rather than commas are used to separate the arguments. Details are in Table 4.6, “SQL String Functions and Operators”. These functions are also implemented using the regular syntax for function invocation. (See Table 4.7, “Other String Functions”.)

Table 4.6. SQL String Functions and Operators

Function	Return Type	Description	Example	Result
string || *string*	text	string concatenation	'Postgre' || 'SQL'	PostgreSQL
bit_length(*string*)	integer	number of bits in string	bit_length('jose')	32
char_length(*string*) or character_length(*string*)	integer	number of characters in string	char_length('jose')	4
lower(*string*)	text	Convert string to lower case.	lower('TOM')	tom
octet_length(*string*)	integer	number of bytes in string	octet_length('jose')	4
position(*substring* in *string*)	integer	location of specified substring	position('om' in 'Thomas')	3

Function	Return Type	Description	Example	Result
substring(*string* [from integer] [for integer])	text	extract substring	substring('Thomas' from 2 for 3)	hom
trim([leading | trailing | both] [*characters*] from *string*)	text	Removes the longest string containing only the *characters* (a space by default) from the beginning/end/both ends of the *string*.	trim(both 'x' from 'xTomxx')	Tom
upper(*string*)	text	Convert string to upper case.	upper('tom')	TOM

Additional string manipulation functions are available and are listed below. Some of them are used internally to implement the SQL-standard string functions listed above.

Table 4.7. Other String Functions

Function	Return Type	Description	Example	Result
ascii(text)	integer	Returns the ASCII code of the first character of the argument.	ascii('x')	120
btrim(*string* text, *trim* text)	text	Remove (trim) the longest string consisting only of characters in *trim* from the start and end of *string*.	btrim('xyxtrimyyx','xy')	trim
chr(integer)	text	Returns the character with the given ASCII code.	chr(65)	A
convert(*string* text, [*src_encoding* name,] *dest_encoding* name)	text	Converts string using *dest_encoding*. The original encoding is specified by *src_encoding*. If *src_encoding* is omitted, database encoding is assumed.	convert('text_in_unicode', 'UNICODE', 'LATIN1')	text_in_unicode represented in ISO 8859-1
initcap(text)	text	Converts first letter of each word (whitespace separated) to upper case.	initcap('hi thomas')	Hi Thomas

Function	Return Type	Description	Example	Result
length(*string*)	integer	length of string	length('jose')	4
lpad(*string* text, *length* integer [, *fill* text])	text	Fills up the *string* to length *length* by prepending the characters *fill* (a space by default). If the *string* is already longer than *length* then it is truncated (on the right).	lpad('hi', 5, 'xy')	xyxhi
ltrim(*string* text, *trim* text)	text	Removes the longest string containing only characters from *trim* from the start of the string.	ltrim('zzzytrim', 'xyz')	trim
pg_client_encoding()	name	Returns current client encoding name.	pg_client_encoding()	SQL_ASCII
repeat(text, integer)	text	Repeat text a number of times.	repeat('Pg', 4)	PgPgPgPg
rpad(*string* text, *length* integer [, *fill* text])	text	Fills up the *string* to length *length* by appending the characters *fill* (a space by default). If the *string* is already longer than *length* then it is truncated.	rpad('hi', 5, 'xy')	hixyx
rtrim(*string* text, *trim* text)	text	Removes the longest string containing only characters from *trim* from the end of the string.	rtrim('trimxxxxxx', 'x')	trim
strpos(*string*, *substring*)	text	Locates specified substring. (same as position(*substring* in *string*), but note the reversed argument order)	strpos('high', 'ig')	2
substr(*string*, *from* [, *count*])	text	Extracts specified substring. (same as substring(*string*	substr('alphabet', 3, 2)	ph

Function	Return Type	Description	Example	Result
		from *from* for *count*))		
to_ascii(text [, *encoding*])	text	Converts text from multibyte encoding to ASCII.	to_ascii('Karel')	Karel
translate(*string* text, *from* text, *to* text)	text	Any character in *string* that matches a character in the *from* set is replaced by the corresponding character in the *to* set.	translate('12345', '14', 'ax')	a23x5
encode(*data* bytea, *type* text)	text	Encodes binary data to ASCII-only representation. Supported types are: 'base64', 'hex', 'escape'.	encode('123\\000\\001', 'base64')	MTIzAAE=
decode(*string* text, *type* text)	bytea	Decodes binary data from *string* previously encoded with encode(). Parameter type is same as in encode().	decode('MTIzAAE=', 'base64')	123\000\001

The to_ascii function supports conversion from LATIN1, LATIN2, WIN1250 (CP1250) only.

4.5. Binary String Functions and Operators

This section describes functions and operators for examining and manipulating binary string values. Strings in this context include values of the type BYTEA.

SQL defines some string functions with a special syntax where certain keywords rather than commas are used to separate the arguments. Details are in Table 4.8, “SQL Binary String Functions and Operators”. Some functions are also implemented using the regular syntax for function invocation. (See Table 4.9, “Other Binary String Functions”.)

Table 4.8. SQL Binary String Functions and Operators

Function	Return Type	Description	Example	Result
string || *string*	bytea	string concatenation	'\\\\Postgre'::bytea || '\\047SQL\\000'::bytea	\\Postgre'SQL\000
octet_length(*string*)	integer	number of bytes in binary string	octet_length('jo\\000se'::bytea)	5
position(*substring* in *string*)	integer	location of specified substring	position('om'::bytea in 'Postgre'::bytea)	3

Function	Return Type	Description	Example	Result
			`in 'Th\\000omas'::bytea)`	
`substring(`*`string`*` [from integer] [for integer])`	bytea	extract substring	`substring('Th\\000omas'::bytea from 2 for 3)`	`h\000o`
`trim([both] `*`characters`*` from `*`string`*`)`	bytea	Removes the longest string containing only the *characters* from the beginning/end/both ends of the *string*.	`trim('\\000'::bytea from '\\000Tom\\000'::bytea)`	`Tom`

Additional binary string manipulation functions are available and are listed below. Some of them are used internally to implement the SQL-standard string functions listed above.

Table 4.9. Other Binary String Functions

Function	Return Type	Description	Example	Result
`btrim(`*`string`*` bytea, `*`trim`*` bytea)`	bytea	Remove (trim) the longest string consisting only of characters in *trim* from the start and end of *string*.	`btrim('\\000trim\\000'::bytea,'\\000'::bytea)`	`trim`
`length(`*`string`*`)`	integer	length of binary string	`length('jo\\000se'::bytea)`	5
`encode(`*`string`*` bytea, `*`type`*` text)`	text	Encodes binary string to ASCII-only representation. Supported types are: 'base64', 'hex', 'escape'.	`encode('123\\000456'::bytea, 'escape')`	`123\000456`
`decode(`*`string`*` text, `*`type`*` text)`	bytea	Decodes binary string from *string* previously encoded with encode(). Parameter type is same as in encode().	`decode('123\\000456', 'escape')`	`123\000456`

4.6. Pattern Matching

There are two separate approaches to pattern matching provided by PostgreSQL: the SQL `LIKE` operator and POSIX-style regular expressions.

Tip

If you have pattern matching needs that go beyond this, or want to make pattern-driven substitutions or translations, consider writing a user-defined function in Perl or Tcl.

4.6.1. Pattern Matching with `LIKE`

```
string LIKE pattern [ ESCAPE escape-character ]
string NOT LIKE pattern [ ESCAPE escape-character ]
```

Every *pattern* defines a set of strings. The `LIKE` expression returns true if the *string* is contained in the set of strings represented by *pattern*. (As expected, the `NOT LIKE` expression returns false if `LIKE` returns true, and vice versa. An equivalent expression is `NOT (`*string* `LIKE` *pattern*`)`.)

If *pattern* does not contain percent signs or underscore, then the pattern only represents the string itself; in that case `LIKE` acts like the equals operator. An underscore (_) in *pattern* stands for (matches) any single character; a percent sign (`%`) matches any string of zero or more characters.

Some examples:

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

`LIKE` pattern matches always cover the entire string. To match a pattern anywhere within a string, the pattern must therefore start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, the respective character in *pattern* must be preceded by the escape character. The default escape character is the backslash but a different one may be selected by using the `ESCAPE` clause. To match the escape character itself, write two escape characters.

Note that the backslash already has a special meaning in string literals, so to write a pattern constant that contains a backslash you must write two backslashes in the query. Thus, writing a pattern that actually matches a literal backslash means writing four backslashes in the query. You can avoid this by selecting a different escape character with `ESCAPE`; then backslash is not special to `LIKE` anymore. (But it is still special to the string literal parser, so you still need two of them.)

It's also possible to select no escape character by writing `ESCAPE ''`. In this case there is no way to turn off the special meaning of underscore and percent signs in the pattern.

The keyword `ILIKE` can be used instead of `LIKE` to make the match case insensitive according to the active locale. This is not in the SQL standard but is a PostgreSQL extension.

The operator `~~` is equivalent to `LIKE`, and `~~*` corresponds to `ILIKE`. There are also `!~~` and `!~~*` operators that represent `NOT LIKE` and `NOT ILIKE`. All of these operators are PostgreSQL-specific.

4.6.2. POSIX Regular Expressions

Table 4.10. Regular Expression Match Operators

Operator	Description	Example
~	Matches regular expression, case sensitive	`'thomas' ~ '.*thomas.*'`

Operator	Description	Example
~*	Matches regular expression, case insensitive	'thomas' ~* '.*Thomas.*'
!~	Does not match regular expression, case sensitive	'thomas' !~ '.*Thomas.*'
!~*	Does not match regular expression, case insensitive	'thomas' !~* '.*vadim.*'

POSIX regular expressions provide a more powerful means for pattern matching than the LIKE function. Many Unix tools such as egrep, sed, or awk use a pattern matching language that is similar to the one described here.

A regular expression is a character sequence that is an abbreviated definition of a set of strings (a *regular set*). A string is said to match a regular expression if it is a member of the regular set described by the regular expression. As with LIKE, pattern characters match string characters exactly unless they are special characters in the regular expression language --- but regular expressions use different special characters than LIKE does. Unlike LIKE patterns, a regular expression is allowed to match anywhere within a string, unless the regular expression is explicitly anchored to the beginning or end of the string.

Regular expressions ("RE"s), as defined in POSIX 1003.2, come in two forms: modern REs (roughly those of egrep; 1003.2 calls these "extended" REs) and obsolete REs (roughly those of ed; 1003.2 "basic" REs). PostgreSQL implements the modern form.

A (modern) RE is one or more non-empty *branches*, separated by |. It matches anything that matches one of the branches.

A branch is one or more *pieces*, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an *atom* possibly followed by a single *, +, ?, or *bound*. An atom followed by * matches a sequence of 0 or more matches of the atom. An atom followed by + matches a sequence of 1 or more matches of the atom. An atom followed by ? matches a sequence of 0 or 1 matches of the atom.

A *bound* is { followed by an unsigned decimal integer, possibly followed by , possibly followed by another unsigned decimal integer, always followed by }. The integers must lie between 0 and RE_DUP_MAX (255) inclusive, and if there are two of them, the first may not exceed the second. An atom followed by a bound containing one integer *i* and no comma matches a sequence of exactly *i* matches of the atom. An atom followed by a bound containing one integer *i* and a comma matches a sequence of *i* or more matches of the atom. An atom followed by a bound containing two integers *i* and *j* matches a sequence of *i* through *j* (inclusive) matches of the atom.

Note

A repetition operator (?, *, +, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow ^ or |.

An *atom* is a regular expression enclosed in () (matching a match for the regular expression), an empty set of () (matching the null string), a *bracket expression* (see below), . (matching any single character), ^ (matching the null string at the beginning of the input string), $ (matching the null string at the end of the input string), a \ followed by one of the characters ^.[$()|*+?{\ (matching that character taken as an ordinary character), a \ followed by any other character (matching that character taken as an ordinary character, as if the \ had not been present), or a single character with no other significance (matching that character). A { followed by a character other than a digit is an ordinary character, not the beginning of a bound. It is illegal to end an RE with \.

Note that the backslash (\) already has a special meaning in string literals, so to write a pattern constant that contains a backslash you must write two backslashes in the query.

A *bracket expression* is a list of characters enclosed in `[]`. It normally matches any single character from the list (but see below). If the list begins with `^`, it matches any single character (but see below) not from the rest of the list. If two characters in the list are separated by `-`, this is shorthand for the full range of characters between those two (inclusive) in the collating sequence, e.g. `[0-9]` in ASCII matches any decimal digit. It is illegal for two ranges to share an endpoint, e.g. `a-c-e`. Ranges are very collating-sequence-dependent, and portable programs should avoid relying on them.

To include a literal `]` in the list, make it the first character (following a possible `^`). To include a literal `-`, make it the first or last character, or the second endpoint of a range. To use a literal `-` as the first endpoint of a range, enclose it in `[.` and `.]` to make it a collating element (see below). With the exception of these and some combinations using `[` (see next paragraphs), all other special characters, including `\`, lose their special significance within a bracket expression.

Within a bracket expression, a collating element (a character, a multiple-character sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in `[.` and `.]` stands for the sequence of characters of that collating element. The sequence is a single element of the bracket expression's list. A bracket expression containing a multiple-character collating element can thus match more than one character, e.g. if the collating sequence includes a `ch` collating element, then the RE `[[.ch.]]*c` matches the first five characters of `chchcc`.

Within a bracket expression, a collating element enclosed in `[=` and `=]` is an equivalence class, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were `[.` and `.]`.) For example, if `o` and `^` are the members of an equivalence class, then `[[=o=]]`, `[[=^=]]`, and `[o^]` are all synonymous. An equivalence class may not be an endpoint of a range.

Within a bracket expression, the name of a character class enclosed in `[:` and `:]` stands for the list of all characters belonging to that class. Standard character class names are: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`. These stand for the character classes defined in ctype. A locale may provide others. A character class may not be used as an endpoint of a range.

There are two special cases of bracket expressions: the bracket expressions `[[:<:]]` and `[[:>:]]` match the null string at the beginning and end of a word respectively. A word is defined as a sequence of word characters which is neither preceded nor followed by word characters. A word character is an alnum character (as defined by ctype) or an underscore. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, it matches the longest. Subexpressions also match the longest possible substrings, subject to the constraint that the whole match be as long as possible, with subexpressions starting earlier in the RE taking priority over ones starting later. Note that higher-level subexpressions thus take priority over their lower-level component subexpressions.

Match lengths are measured in characters, not collating elements. A null string is considered longer than no match at all. For example, `bb*` matches the three middle characters of `abbbc`, `(wee|week)(knights|nights)` matches all ten characters of `weeknights`, when `(.*).*` is matched against `abc` the parenthesized subexpression matches all three characters, and when `(a*)*` is matched against `bc` both the whole RE and the parenthesized subexpression match the null string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a

bracket expression, it is effectively transformed into a bracket expression containing both cases, e.g. `x` becomes `[xX]`. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, so that (e.g.) `[x]` becomes `[xX]` and `[^x]` becomes `[^xX]`.

There is no particular limit on the length of REs, except insofar as memory is limited. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. Bounded repetitions are implemented by macro expansion, which is costly in time and space if counts are large or bounded repetitions are nested. An RE like, say, `((((a{1,100}){1,100}){1,100}){1,100}){1,100}` will (eventually) run almost any existing machine out of swap space. [1]

4.7. Data Type Formatting Functions

Author

Written by Karel Zak (<zakkr@zf.jcu.cz>) on 2000-01-24

The PostgreSQL formatting functions provide a powerful set of tools for converting various data types (date/time, integer, floating point, numeric) to formatted strings and for converting from formatted strings to specific data types. These functions all follow a common calling convention: the first argument is the value to be formatted and the second argument is a template that defines the output or input format.

Table 4.11. Formatting Functions

Function	Returns	Description	Example
`to_char(timestamp, text)`	`text`	convert time stamp to string	`to_char(timestamp 'now','HH12:MI:SS')`
`to_char(interval, text)`	`text`	convert interval to string	`to_char(interval '15h 2m 12s','HH24:MI:SS')`
`to_char(int, text)`	`text`	convert int4/int8 to string	`to_char(125, '999')`
`to_char(double precision, text)`	`text`	convert real/double precision to string	`to_char(125.8, '999D9')`
`to_char(numeric, text)`	`text`	convert numeric to string	`to_char(numeric '-125.8', '999D99S')`
`to_date(text, text)`	`date`	convert string to date	`to_date('05 Dec 2000', 'DD Mon YYYY')`
`to_timestamp(text, text)`	`timestamp`	convert string to time stamp	`to_timestamp('05 Dec 2000', 'DD Mon YYYY')`
`to_number(text, text)`	`numeric`	convert string to numeric	`to_number('12,454.8-', '99G999D9S')`

In an output template string, there are certain patterns that are recognized and replaced with appropriately-formatted data from the value to be formatted. Any text that is not a template pattern is simply copied verbatim. Similarly, in an input template string, template patterns identify the parts of the input data string to be looked at and the values to be found there.

[1] This was written in 1994, mind you. The numbers have probably changed, but the problem persists.

Table 4.12. Template patterns for date/time conversions

Pattern	Description
HH	hour of day (01-12)
HH12	hour of day (01-12)
HH24	hour of day (00-23)
MI	minute (00-59)
SS	second (00-59)
MS	millisecond (000-999)
US	microsecond (000000-999999)
SSSS	seconds past midnight (0-86399)
AM or A.M. or PM or P.M.	meridian indicator (upper case)
am or a.m. or pm or p.m.	meridian indicator (lower case)
Y,YYY	year (4 and more digits) with comma
YYYY	year (4 and more digits)
YYY	last 3 digits of year
YY	last 2 digits of year
Y	last digit of year
BC or B.C. or AD or A.D.	era indicator (upper case)
bc or b.c. or ad or a.d.	era indicator (lower case)
MONTH	full upper case month name (blank-padded to 9 chars)
Month	full mixed case month name (blank-padded to 9 chars)
month	full lower case month name (blank-padded to 9 chars)
MON	abbreviated upper case month name (3 chars)
Mon	abbreviated mixed case month name (3 chars)
mon	abbreviated lower case month name (3 chars)
MM	month number (01-12)
DAY	full upper case day name (blank-padded to 9 chars)
Day	full mixed case day name (blank-padded to 9 chars)
day	full lower case day name (blank-padded to 9 chars)
DY	abbreviated upper case day name (3 chars)
Dy	abbreviated mixed case day name (3 chars)
dy	abbreviated lower case day name (3 chars)
DDD	day of year (001-366)
DD	day of month (01-31)
D	day of week (1-7; SUN=1)
W	week of month (1-5) where first week start on the first day of the month

Pattern	Description
WW	week number of year (1-53) where first week start on the first day of the year
IW	ISO week number of year (The first Thursday of the new year is in week 1.)
CC	century (2 digits)
J	Julian Day (days since January 1, 4712 BC)
Q	quarter
RM	month in Roman Numerals (I-XII; I=January) - upper case
rm	month in Roman Numerals (I-XII; I=January) - lower case
TZ	timezone name - upper case
tz	timezone name - lower case

Certain modifiers may be applied to any template pattern to alter its behavior. For example, "FMMonth" is the "Month" pattern with the "FM" prefix.

Table 4.13. Template pattern modifiers for date/time conversions

Modifier	Description	Example
FM prefix	fill mode (suppress padding blanks and zeroes)	FMMonth
TH suffix	add upper-case ordinal number suffix	DDTH
th suffix	add lower-case ordinal number suffix	DDth
FX prefix	Fixed format global option (see below)	FX Month DD Day
SP suffix	spell mode (not yet implemented)	DDSP

Usage notes:

- FM suppresses leading zeroes or trailing blanks that would otherwise be added to make the output of a pattern be fixed-width.
- to_timestamp and to_date skip multiple blank spaces in the input string if the FX option is not used. FX must be specified as the first item in the template; for example to_timestamp('2000 JUN','YYYY MON') is right, but to_timestamp('2000 JUN','FXYYYY MON') returns an error, because to_timestamp expects one blank space only.
- If a backslash ("\") is desired in a string constant, a double backslash ("\\") must be entered; for example '\\HH\\MI\\SS'. This is true for any string constant in PostgreSQL.
- Ordinary text is allowed in to_char templates and will be output literally. You can put a substring in double quotes to force it to be interpreted as literal text even if it contains pattern keywords. For example, in '"Hello Year: "YYYY', the YYYY will be replaced by year data, but the single Y will not be.
- If you want to have a double quote in the output you must precede it with a backslash, for example '\\"YYYY Month\\"'.

- `YYYY` conversion from string to `timestamp` or `date` is restricted if you use a year with more than 4 digits. You must use some non-digit character or template after `YYYY`, otherwise the year is always interpreted as 4 digits. For example (with year 20000): `to_date('200001131', 'YYYYMMDD')` will be interpreted as a 4-digit year; better is to use a non-digit separator after the year, like `to_date('20000-1131', 'YYYY-MMDD')` or `to_date('20000Nov31', 'YYYYMonDD')`.

- Millisecond `MS` and microsecond `US` values in a conversion from string to time stamp are used as part of the seconds after the decimal point. For example `to_timestamp('12:3', 'SS:MS')` is not 3 milliseconds, but 300, because the conversion counts it as 12 + 0.3. This means for the format `SS:MS`, the input values `12:3`, `12:30`, and `12:300` specify the same number of milliseconds. To get three milliseconds, one must use `12:003`, which the conversion counts as 12 + 0.003 = 12.003 seconds.

  Here is a more complex example: `to_timestamp('15:12:02.020.001230','HH:MI:SS.MS.US')` is 15 hours, 12 minutes, and 2 seconds + 20 milliseconds + 1230 microseconds = 2.021230 seconds.

Table 4.14. Template patterns for numeric conversions

Pattern	Description
`9`	value with the specified number of digits
`0`	value with leading zeros
`.` (period)	decimal point
`,` (comma)	group (thousand) separator
`PR`	negative value in angle brackets
`S`	negative value with minus sign (uses locale)
`L`	currency symbol (uses locale)
`D`	decimal point (uses locale)
`G`	group separator (uses locale)
`MI`	minus sign in specified position (if number < 0)
`PL`	plus sign in specified position (if number > 0)
`SG`	plus/minus sign in specified position
`RN`	roman numeral (input between 1 and 3999)
`TH` or `th`	convert to ordinal number
`V`	shift *n* digits (see notes)
`EEEE`	scientific notation (not implemented yet)

Usage notes:

- A sign formatted using `SG`, `PL`, or `MI` is not an anchor in the number; for example, `to_char(-12, 'S9999')` produces `' -12'`, but `to_char(-12, 'MI9999')` produces `'- 12'`. The Oracle implementation does not allow the use of `MI` ahead of `9`, but rather requires that `9` precede `MI`.

- `9` specifies a value with the same number of digits as there are `9`s. If a digit is not available use blank space.

- `TH` does not convert values less than zero and does not convert decimal numbers.

- `PL`, `SG`, and `TH` are PostgreSQL extensions.

- `V` effectively multiplies the input values by `10^n`, where *n* is the number of digits following `V`. `to_char` does not support the use of `V` combined with a decimal point. (E.g., `99.9V99` is not allowed.)

Table 4.15. `to_char` Examples

Input	Output
`to_char(now(),'Day, DD HH12:MI:SS')`	`'Tuesday , 06 05:39:18'`
`to_char(now(),'FMDay, FMDD HH12:MI:SS')`	`'Tuesday, 6 05:39:18'`
`to_char(-0.1,'99.99')`	`' -.10'`
`to_char(-0.1,'FM9.99')`	`'-.1'`
`to_char(0.1,'0.9')`	`' 0.1'`
`to_char(12,'9990999.9')`	`' 0012.0'`
`to_char(12,'FM9990999.9')`	`'0012'`
`to_char(485,'999')`	`' 485'`
`to_char(-485,'999')`	`'-485'`
`to_char(485,'9 9 9')`	`' 4 8 5'`
`to_char(1485,'9,999')`	`' 1,485'`
`to_char(1485,'9G999')`	`' 1 485'`
`to_char(148.5,'999.999')`	`' 148.500'`
`to_char(148.5,'999D999')`	`' 148,500'`
`to_char(3148.5,'9G999D999')`	`' 3 148,500'`
`to_char(-485,'999S')`	`'485-'`
`to_char(-485,'999MI')`	`'485-'`
`to_char(485,'999MI')`	`'485'`
`to_char(485,'PL999')`	`'+485'`
`to_char(485,'SG999')`	`'+485'`
`to_char(-485,'SG999')`	`'-485'`
`to_char(-485,'9SG99')`	`'4-85'`
`to_char(-485,'999PR')`	`'<485>'`
`to_char(485,'L999')`	`'DM 485`
`to_char(485,'RN')`	`' CDLXXXV'`
`to_char(485,'FMRN')`	`'CDLXXXV'`
`to_char(5.2,'FMRN')`	`V`
`to_char(482,'999th')`	`' 482nd'`
`to_char(485, '"Good number:"999')`	`'Good number: 485'`
`to_char(485.8,'"Pre:"999" Post:" .999')`	`'Pre: 485 Post: .800'`
`to_char(12,'99V999')`	`' 12000'`
`to_char(12.4,'99V999')`	`' 12400'`

Input	Output
`to_char(12.45, '99V9')`	`' 125'`

4.8. Date/Time Functions and Operators

Table 4.17, “Date/Time Functions” shows the available functions for date/time value processing. Table 4.16, “Date/Time Operators” illustrates the behaviors of the basic arithmetic operators (+, *, etc.). For formatting functions, refer to Section 4.7, “Data Type Formatting Functions”. You should be familiar with the background information on date/time data types (see Section 3.5, “Date/Time Types”).

The date/time operators described below behave similarly for types involving time zones as well as those without.

Table 4.16. Date/Time Operators

Name	Example	Result
+	`timestamp '2001-09-28 01:00' + interval '23 hours'`	`timestamp '2001-09-29 00:00'`
+	`date '2001-09-28' + interval '1 hour'`	`timestamp '2001-09-28 01:00'`
+	`time '01:00' + interval '3 hours'`	`time '04:00'`
-	`timestamp '2001-09-28 23:00' - interval '23 hours'`	`timestamp '2001-09-28'`
-	`date '2001-09-28' - interval '1 hour'`	`timestamp '2001-09-27 23:00'`
-	`time '05:00' - interval '2 hours'`	`time '03:00'`
-	`interval '2 hours' - time '05:00'`	`time '03:00:00'`
*	`interval '1 hour' * int '3'`	`interval '03:00'`
/	`interval '1 hour' / int '3'`	`interval '00:20'`

The date/time functions are summarized below, with additional details in subsequent sections.

Table 4.17. Date/Time Functions

Name	Return Type	Description	Example	Result
`age(timestamp)`	`interval`	Subtract from today	`age(timestamp '1957-06-13')`	`43 years 8 mons 3 days`
`age(timestamp, timestamp)`	`interval`	Subtract arguments	`age('2001-04-10', timestamp '1957-06-13')`	`43 years 9 mons 27 days`
`current_date`	`date`	Today's date; see below		
`current_time`	`time`	Time of day; see below		

Name	Return Type	Description	Example	Result
current_timestamp	timestamp	Date and time; see below		
date_part(text, timestamp)	double precision	Get subfield (equivalent to extract); see also below	date_part('hour', timestamp '2001-02-16 20:38:40')	20
date_part(text, interval)	double precision	Get subfield (equivalent to extract); see also below	date_part('month', interval '2 years 3 months')	3
date_trunc(text, timestamp)	timestamp	Truncate to specified precision; see also below	date_trunc('hour', timestamp '2001-02-16 20:38:40')	2001-02-16 20:00:00+00
extract(*field* from timestamp)	double precision	Get subfield; see also below	extract(hour from timestamp '2001-02-16 20:38:40')	20
extract(*field* from interval)	double precision	Get subfield; see also below	extract(month from interval '2 years 3 months')	3
isfinite(timestamp)	boolean	Test for finite time stamp (neither invalid nor infinity)	isfinite(timestamp '2001-02-16 21:28:30')	true
isfinite(interval)	boolean	Test for finite interval	isfinite(interval '4 hours')	true
now()	timestamp	Current date and time (equivalent to current_timestamp); see below		
timeofday()	text	Current date and time; see below	timeofday()	Wed Feb 21 17:01:13.000126 2001 EST
timestamp(date)	timestamp	date to timestamp	timestamp(date '2000-12-25')	2000-12-25 00:00:00
timestamp(date, time)	timestamp	date and time to timestamp	timestamp(date '1998-02-24', time '23:07')	1998-02-24 23:07:00

4.8.1. EXTRACT, date_part

```
EXTRACT (field FROM source)
```

The extract function retrieves sub-fields from date/time values, such as year or hour. *source* is a value expression that evaluates to type timestamp or interval. (Expressions of type date or time will be cast to timestamp and can therefore be used as well.) *field* is an identifier or string that

selects what field to extract from the source value. The `extract` function returns values of type `double precision`. The following are valid values:

`century`

The year field divided by 100

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20
```

Note that the result for the century field is simply the year field divided by 100, and not the conventional definition which puts most years in the 1900's in the twentieth century.

`day`

The day (of the month) field (1 - 31)

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 16
```

`decade`

The year field divided by 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 200
```

`dow`

The day of the week (0 - 6; Sunday is 0) (for `timestamp` values only)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 5
```

`doy`

The day of the year (1 - 365/366) (for `timestamp` values only)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 47
```

`epoch`

For `date` and `timestamp` values, the number of seconds since 1970-01-01 00:00:00-00 (Result may be negative.); for `interval` values, the total number of seconds in the interval

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 982352320

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Result: 442800
```

`hour`

The hour field (0 - 23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20
```

`microseconds`

The seconds field, including fractional parts, multiplied by 1 000 000. Note that this includes full seconds.

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Result: 28500000
```

`millennium`

The year field divided by 1000

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2
```

Note that the result for the millennium field is simply the year field divided by 1000, and not the conventional definition which puts years in the 1900's in the second millennium.

`milliseconds`

The seconds field, including fractional parts, multiplied by 1000. Note that this includes full seconds.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Result: 28500
```

`minute`

The minutes field (0 - 59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 38
```

`month`

For `timestamp` values, the number of the month within the year (1 - 12) ; for `interval` values the number of months, modulo 12 (0 - 11)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Result: 3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Result: 1
```

`quarter`

The quarter of the year (1 - 4) that the day is in (for `timestamp` values only)

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 1
```

`second`

The seconds field, including fractional parts (0 - 59[2])

[2] 60 if leap seconds are implemented by the operating system

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Result: 28.5
```

`timezone_hour`

The hour component of the time zone offset.

`timezone_minute`

The minute component of the time zone offset.

`week`

From a `timestamp` value, calculate the number of the week of the year that the day is in. By definition (ISO 8601), the first week of a year contains January 4 of that year. (The ISO week starts on Monday.) In other words, the first Thursday of a year is in week 1 of that year.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 7
```

`year`

The year field

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2001
```

The `extract` function is primarily intended for computational processing. For formatting date/time values for display, see Section 4.7, “Data Type Formatting Functions”.

The `date_part` function is modeled on the traditional Ingres equivalent to the SQL-function `extract`:

`date_part('field', source)`

Note that here the *`field`* value needs to be a string. The valid field values for `date_part` are the same as for `extract`.

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Result: 16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Result: 4
```

4.8.2. date_trunc

The function `date_trunc` is conceptually similar to the `trunc` function for numbers.

`date_trunc('field', source)`

`source` is a value expression of type `timestamp` (values of type `date` and `time` are cast automatically). *`field`* selects to which precision to truncate the time stamp value. The return value is of type `timestamp` with all fields that are less than the selected one set to zero (or one, for day and month).

Valid values for *field* are:

microseconds
milliseconds
second
minute
hour
day
month
year
decade
century
millennium

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-02-16 20:00:00+00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-01-01 00:00:00+00
```

4.8.3. Current Date/Time

The following functions are available to obtain the current date and/or time:

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME ( precision )
CURRENT_TIMESTAMP ( precision )
```

`CURRENT_TIME` and `CURRENT_TIMESTAMP` can optionally be given a precision parameter, which causes the result to be rounded to that many fractional digits. Without a precision parameter, the result is given to full available precision.

Note

Prior to PostgreSQL 7.2, the precision parameters were unimplemented, and the result was always given in integer seconds.

Note

The SQL99 standard requires these functions to be written without any parentheses, unless a precision parameter is given. As of PostgreSQL 7.2, an empty pair of parentheses can be written, but this is deprecated and may be removed in a future release.

```
SELECT CURRENT_TIME;
14:39:53.662522-05

SELECT CURRENT_DATE;
2001-12-23

SELECT CURRENT_TIMESTAMP;
2001-12-23 14:39:53.662522-05
```

```
SELECT CURRENT_TIMESTAMP(2);
2001-12-23 14:39:53.66-05
```

The function `now()` is the traditional PostgreSQL equivalent to `CURRENT_TIMESTAMP`.

There is also `timeofday()`, which for historical reasons returns a text string rather than a `timestamp` value:

```
SELECT timeofday();
 Sat Feb 17 19:07:32.000126 2001 EST
```

It is quite important to realize that `CURRENT_TIMESTAMP` and related functions all return the time as of the start of the current transaction; their values do not increment while a transaction is running. But `timeofday()` returns the actual current time.

All the date/time data types also accept the special literal value `now` to specify the current date and time. Thus, the following three all return the same result:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';
```

Note

You do not want to use the third form when specifying a DEFAULT value while creating a table. The system will convert `now` to a `timestamp` as soon as the constant is parsed, so that when the default value is needed, the time of the table creation would be used! The first two forms will not be evaluated until the default value is used, because they are function calls. Thus they will give the desired behavior of defaulting to the time of row insertion.

4.9. Geometric Functions and Operators

The geometric types `point`, `box`, `lseg`, `line`, `path`, `polygon`, and `circle` have a large set of native support functions and operators.

Table 4.18. Geometric Operators

Operator	Description	Usage
+	Translation	`box '((0,0),(1,1))' + point '(2.0,0)'`
-	Translation	`box '((0,0),(1,1))' - point '(2.0,0)'`
*	Scaling/rotation	`box '((0,0),(1,1))' * point '(2.0,0)'`
/	Scaling/rotation	`box '((0,0),(2,2))' / point '(2.0,0)'`
#	Intersection	`'((1,-1),(-1,1))' # '((1,1),(-1,-1))'`
#	Number of points in polygon	`# '((1,0),(0,1),(-1,0))'`

Operator	Description	Usage
##	Point of closest proximity	`point '(0,0)' ## lseg '((2,0),(0,2))'`
&&	Overlaps?	`box '((0,0),(1,1))' && box '((0,0),(2,2))'`
&<	Overlaps to left?	`box '((0,0),(1,1))' &< box '((0,0),(2,2))'`
&>	Overlaps to right?	`box '((0,0),(3,3))' &> box '((0,0),(2,2))'`
<->	Distance between	`circle '((0,0),1)' <-> circle '((5,0),1)'`
<<	Left of?	`circle '((0,0),1)' << circle '((5,0),1)'`
<^	Is below?	`circle '((0,0),1)' <^ circle '((0,5),1)'`
>>	Is right of?	`circle '((5,0),1)' >> circle '((0,0),1)'`
>^	Is above?	`circle '((0,5),1)' >^ circle '((0,0),1)'`
?#	Intersects or overlaps	`lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'`
?-	Is horizontal?	`point '(1,0)' ?- point '(0,0)'`
?-|	Is perpendicular?	`lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'`
@-@	Length or circumference	`@-@ path '((0,0),(1,0))'`
?|	Is vertical?	`point '(0,1)' ?| point '(0,0)'`
?||	Is parallel?	`lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'`
@	Contained or on	`point '(1,1)' @ circle '((0,0),2)'`
@@	Center of	`@@ circle '((0,0),10)'`
~=	Same as	`polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'`

Table 4.19. Geometric Functions

Function	Returns	Description	Example
`area`(object)	`double precision`	area of item	`area(box '((0,0),(1,1))')`

Function	Returns	Description	Example
box(box, box)	box	intersection box	box(box '((0,0),(1,1))',box '((0.5,0.5),(2,2))')
center(object)	point	center of item	center(box '((0,0),(1,2))')
diameter(circle)	double precision	diameter of circle	diameter(circle '((0,0),2.0)')
height(box)	double precision	vertical size of box	height(box '((0,0),(1,1))')
isclosed(path)	boolean	a closed path?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	an open path?	isopen(path '[(0,0),(1,1),(2,0)]')
length(object)	double precision	length of item	length(path '((-1,0),(1,0))')
pclose(path)	path	convert path to closed	popen(path '[(0,0),(1,1),(2,0)]')
npoint(path)	integer	number of points	npoints(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	convert path to open path	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	radius of circle	radius(circle '((0,0),2.0)')
width(box)	double precision	horizontal size	width(box '((0,0),(1,1))')

Table 4.20. Geometric Type Conversion Functions

Function	Returns	Description	Example
box(circle)	box	circle to box	box(circle '((0,0),2.0)')
box(point, point)	box	points to box	box(point '(0,0)', point '(1,1)')
box(polygon)	box	polygon to box	box(polygon '((0,0),(1,1),(2,0))')
circle(box)	circle	to circle	circle(box '((0,0),(1,1))')

Function	Returns	Description	Example
`circle(point, double precision)`	`circle`	point to circle	`circle(point '(0,0)', 2.0)`
`lseg(box)`	`lseg`	box diagonal to lseg	`lseg(box '((-1,0),(1,0))')`
`lseg(point, point)`	`lseg`	points to lseg	`lseg(point '(-1,0)', point '(1,0)')`
`path(polygon)`	`point`	polygon to path	`path(polygon '((0,0),(1,1),(2,0))')`
`point(circle)`	`point`	center	`point(circle '((0,0),2.0)')`
`point(lseg, lseg)`	`point`	intersection	`point(lseg '((-1,0),(1,0))', lseg '((-2,-2),(2,2))')`
`point(polygon)`	`point`	center	`point(polygon '((0,0),(1,1),(2,0))')`
`polygon(box)`	`polygon`	12 point polygon	`polygon(box '((0,0),(1,1))')`
`polygon(circle)`	`polygon`	12-point polygon	`polygon(circle '((0,0),2.0)')`
`polygon(`*`npts`*`, circle)`	`polygon`	*`npts`* polygon	`polygon(12, circle '((0,0),2.0)')`
`polygon(path)`	`polygon`	path to polygon	`polygon(path '((0,0),(1,1),(2,0))')`

4.10. Network Address Type Functions

Table 4.21. `cidr` and `inet` Operators

Operator	Description	Usage
<	Less than	`inet '192.168.1.5' < inet '192.168.1.6'`
<=	Less than or equal	`inet '192.168.1.5' <= inet '192.168.1.5'`
=	Equals	`inet '192.168.1.5' = inet '192.168.1.5'`
>=	Greater or equal	`inet '192.168.1.5' >= inet '192.168.1.5'`

Operator	Description	Usage
>	Greater	inet '192.168.1.5' > inet '192.168.1.4'
<>	Not equal	inet '192.168.1.5' <> inet '192.168.1.4'
<<	is contained within	inet '192.168.1.5' << inet '192.168.1/24'
<<=	is contained within or equals	inet '192.168.1/24' <<= inet '192.168.1/24'
>>	contains	inet'192.168.1/24' >> inet '192.168.1.5'
>>=	contains or equals	inet '192.168.1/24' >>= inet '192.168.1/24'

All of the operators for inet can be applied to cidr values as well. The operators <<, <<=, >>, >>= test for subnet inclusion: they consider only the network parts of the two addresses, ignoring any host part, and determine whether one network part is identical to or a subnet of the other.

Table 4.22. cidr and inet Functions

Function	Returns	Description	Example	Result
broadcast(inet)	inet	broadcast address for network	broadcast('192.168.1.5/24')	192.168.1.255/24
host(inet)	text	extract IP address as text	host('192.168.1.5/24')	192.168.1.5
masklen(inet)	integer	extract netmask length	masklen('192.168.1.5/24')	24
set_masklen(inet, integer)	inet	set netmask length for inet value	set_masklen('192.168.1.5/24', 16)	192.168.1.5/16
netmask(inet)	inet	construct netmask for network	netmask('192.168.1.5/24')	255.255.255.0
network(inet)	cidr	extract network part of address	network('192.168.1.5/24')	192.168.1.0/24
text(inet)	text	extract IP address and masklen as text	text(inet '192.168.1.5')	192.168.1.5/32
abbrev(inet)	text	extract abbreviated display as text	abbrev(cidr '10.1.0.0/16')	10.1/16

All of the functions for inet can be applied to cidr values as well. The host(), text(), and abbrev() functions are primarily intended to offer alternative display formats. You can cast a text field to inet using normal casting syntax: inet(expression) or colname::inet.

Table 4.23. macaddr Functions

Function	Returns	Description	Example	Result
trunc(macaddr)	macaddr	set last 3 bytes to zero	trunc(macaddr '12:34:56:78:90:ab')	12:34:56:00:00:00

The function `trunc(macaddr)` returns a MAC address with the last 3 bytes set to 0. This can be used to associate the remaining prefix with a manufacturer. The directory `contrib/mac` in the source distribution contains some utilities to create and maintain such an association table.

The `macaddr` type also supports the standard relational operators (>, <=, etc.) for lexicographical ordering.

4.11. Sequence-Manipulation Functions

Table 4.24. Sequence Functions

Function	Returns	Description
`nextval(text)`	`bigint`	Advance sequence and return new value
`currval(text)`	`bigint`	Return value most recently obtained with `nextval`
`setval(text,bigint)`	`bigint`	Set sequence's current value
`setval(text,bigint,boolean)`	`bigint`	Set sequence's current value and `is_called` flag

This section describes PostgreSQL's functions for operating on *sequence objects*. Sequence objects (also called sequence generators or just sequences) are special single-row tables created with `CREATE SEQUENCE`. A sequence object is usually used to generate unique identifiers for rows of a table. The sequence functions provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

For largely historical reasons, the sequence to be operated on by a sequence-function call is specified by a text-string argument. To achieve some compatibility with the handling of ordinary SQL names, the sequence functions convert their argument to lower case unless the string is double-quoted. Thus

```
nextval('foo')      operates on sequence foo
nextval('FOO')      operates on sequence foo
nextval('"Foo"')    operates on sequence Foo
```

Of course, the text argument can be the result of an expression, not only a simple literal, which is occasionally useful.

The available sequence functions are:

`nextval`

Advance the sequence object to its next value and return that value. This is done atomically: even if multiple server processes execute `nextval` concurrently, each will safely receive a distinct sequence value.

`currval`

Return the value most recently obtained by `nextval` for this sequence in the current server process. (An error is reported if `nextval` has never been called for this sequence in this process.) Notice that because this is returning a process-local value, it gives a predictable answer even if other server processes are executing `nextval` meanwhile.

`setval`

Reset the sequence object's counter value. The two-parameter form sets the sequence's `last_value` field to the specified value and sets its `is_called` field to `true`, meaning that the next `nextval` will advance the sequence before returning a value. In the three-parameter form, `is_called` may be set either `true` or `false`. If it's set to `false`, the next `nextval` will return exactly the specified value, and sequence advancement commences with the following `nextval`. For example,

```
SELECT setval('foo', 42);           Next nextval() will return 43
SELECT setval('foo', 42, true);     Same as above
SELECT setval('foo', 42, false);    Next nextval() will return 42
```

The result returned by `setval` is just the value of its second argument.

Important

To avoid blocking of concurrent transactions that obtain numbers from the same sequence, a `nextval` operation is never rolled back; that is, once a value has been fetched it is considered used, even if the transaction that did the `nextval` later aborts. This means that aborted transactions may leave unused “holes” in the sequence of assigned values. `setval` operations are never rolled back, either.

If a sequence object has been created with default parameters, `nextval()` calls on it will return successive values beginning with one. Other behaviors can be obtained by using special parameters in the `CREATE SEQUENCE` command; see its command reference page for more information.

4.12. Conditional Expressions

This section describes the SQL-compliant conditional expressions available in PostgreSQL.

Tip

If your needs go beyond the capabilities of these conditional expressions you might want to consider writing a stored procedure in a more expressive programming language.

CASE

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

The SQL `CASE` expression is a generic conditional expression, similar to if/else statements in other languages. `CASE` clauses can be used wherever an expression is valid. *`condition`* is an expression that returns a `boolean` result. If the result is true then the value of the `CASE` expression is *`result`*. If the result is false any subsequent `WHEN` clauses are searched in the same manner. If no `WHEN` *`condition`* is true then the value of the case expression is the *`result`* in the `ELSE` clause. If the `ELSE` clause is omitted and no condition matches, the result is NULL.

An example:

```
=> SELECT * FROM test;
 a
---
```

```
 1
 2
 3

=> SELECT a,
          CASE WHEN a=1 THEN 'one'
               WHEN a=2 THEN 'two'
               ELSE 'other'
          END
    FROM test;
 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

The data types of all the *result* expressions must be coercible to a single output type. See Section 5.6, "UNION and CASE Constructs" for more detail.

```
CASE expression
    WHEN value THEN result
    [WHEN ...]
    [ELSE result]
END
```

This "simple" CASE expression is a specialized variant of the general form above. The *expression* is computed and compared to all the *value*s in the WHEN clauses until one is found that is equal. If no match is found, the *result* in the ELSE clause (or NULL) is returned. This is similar to the switch statement in C.

The example above can be written using the simple CASE syntax:

```
=> SELECT a,
          CASE a WHEN 1 THEN 'one'
                 WHEN 2 THEN 'two'
                 ELSE 'other'
          END
    FROM test;
 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

COALESCE

```
COALESCE(value[, ...])
```

The COALESCE function returns the first of its arguments that is not NULL. This is often useful to substitute a default value for NULL values when data is retrieved for display, for example:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

NULLIF

```
NULLIF(value1, value2)
```

The NULLIF function returns NULL if and only if *value1* and *value2* are equal. Otherwise it returns *value1*. This can be used to perform the inverse operation of the COALESCE example given above:

```
SELECT NULLIF(value, '(none)') ...
```

Tip

COALESCE and NULLIF are just shorthand for CASE expressions. They are actually converted into CASE expressions at a very early stage of processing, and subsequent processing thinks it is dealing with CASE. Thus an incorrect COALESCE or NULLIF usage may draw an error message that refers to CASE.

4.13. Miscellaneous Functions

Table 4.25. Session Information Functions

Name	Return Type	Description
current_user	name	user name of current execution context
session_user	name	session user name
user	name	equivalent to current_user

The session_user is the user that initiated a database connection; it is fixed for the duration of that connection. The current_user is the user identifier that is applicable for permission checking. Currently it is always equal to the session user, but in the future there might be “setuid” functions and other facilities to allow the current user to change temporarily. In Unix parlance, the session user is the “real user” and the current user is the “effective user”.

Note that these functions have special syntactic status in SQL: they must be called without trailing parentheses.

Deprecated

The function getpgusername() is an obsolete equivalent of current_user.

Table 4.26. System Information Functions

Name	Return Type	Description
version	text	PostgreSQL version information

version() returns a string describing the PostgreSQL server's version.

Table 4.27. Access Privilege Inquiry Functions

Name	Return Type	Description
has_table_privilege(*user*, *table*, *access*)	boolean	does user have access to table

Name	Return Type	Description
has_table_privilege(*table*, *access*)	boolean	does current user have access to table

has_table_privilege determines whether a user can access a table in a particular way. The user can be specified by name or by ID (pg_user.usesysid), or if the argument is omitted current_user is assumed. The table can be specified by name or by OID. (Thus, there are actually six variants of has_table_privilege, which can be distinguished by the number and types of their arguments.) The desired access type is specified by a text string, which must evaluate to one of the values SELECT, INSERT, UPDATE, DELETE, RULE, REFERENCES, or TRIGGER. (Case of the string is not significant, however.)

Table 4.28. Catalog Information Functions

Name	Return Type	Description
pg_get_viewdef(*viewname*)	text	Get CREATE VIEW command for view
pg_get_ruledef(*rulename*)	text	Get CREATE RULE command for rule
pg_get_indexdef(*indexOID*)	text	Get CREATE INDEX command for index
pg_get_userbyid(*userid*)	name	Get user name given ID

These functions extract information from the system catalogs. pg_get_viewdef(), pg_get_ruledef(), and pg_get_indexdef() respectively reconstruct the creating command for a view, rule, or index. (Note that this is a decompiled reconstruction, not the verbatim text of the command.) pg_get_userbyid() extracts a user's name given a usesysid value.

Table 4.29. Comment Information Functions

Name	Return Type	Description
obj_description(*objectOID*, *tablename*)	text	Get comment for a database object
obj_description(*objectOID*)	text	Get comment for a database object (*deprecated*)
col_description(*tableOID*, *columnnumber*)	text	Get comment for a table column

These functions extract comments previously stored with the COMMENT command. NULL is returned if no comment can be found matching the specified parameters.

The two-parameter form of obj_description() returns the comment for a database object specified by its OID and the name of the containing system catalog. For example, obj_description(123456,'pg_class') would retrieve the comment for a table with OID 123456. The one-parameter form of obj_description() requires only the object OID. It is now deprecated since there is no guarantee that OIDs are unique across different system catalogs; therefore, the wrong comment could be returned.

`col_description()` returns the comment for a table column, which is specified by the OID of its table and its column number. `obj_description()` cannot be used for table columns since columns do not have OIDs of their own.

4.14. Aggregate Functions

Author

Written by Isaac Wilcox <isaac@azartmedia.com> on 2000-06-16

Aggregate functions compute a single result value from a set of input values. The special syntax considerations for aggregate functions are explained in Section 1.3.5, “Aggregate Expressions”. Consult the *PostgreSQL Tutorial* for additional introductory information.

Table 4.30. Aggregate Functions

Function	Description	Notes
AVG(*expression*)	the average (arithmetic mean) of all input values	Finding the average value is available on the following data types: `smallint`, `integer`, `bigint`, `real`, `double precision`, `numeric`, `interval`. The result is of type `numeric` for any integer type input, `double precision` for floating-point input, otherwise the same as the input data type.
`count`(*)	number of input values	The return value is of type `bigint`.
`count`(*expression*)	Counts the input values for which the value of *expression* is not NULL.	The return value is of type `bigint`.
`max`(*expression*)	the maximum value of *expression* across all input values	Available for all numeric, string, and date/time types. The result has the same type as the input expression.
`min`(*expression*)	the minimum value of *expression* across all input values	Available for all numeric, string, and date/time types. The result has the same type as the input expression.
`stddev`(*expression*)	the sample standard deviation of the input values	Finding the standard deviation is available on the following data types: `smallint`, `integer`, `bigint`, `real`, `double precision`, `numeric`. The result is of type `double precision` for floating-point input, otherwise `numeric`.
`sum`(*expression*)	sum of *expression* across all input values	Summation is available on the following data types: `smallint`,

Function	Description	Notes
		integer, bigint, real, double precision, numeric, interval. The result is of type bigint for smallint or integer input, numeric for bigint input, double precision for floating-point input, otherwise the same as the input data type.
variance(*expression*)	the sample variance of the input values	The variance is the square of the standard deviation. The supported data types and result types are the same as for standard deviation.

It should be noted that except for COUNT, these functions return NULL when no rows are selected. In particular, SUM of no rows returns NULL, not zero as one might expect. COALESCE may be used to substitute zero for NULL when necessary.

4.15. Subquery Expressions

This section describes the SQL-compliant subquery expressions available in PostgreSQL. All of the expression forms documented in this section return Boolean (true/false) results.

EXISTS

```
EXISTS ( subquery )
```

The argument of EXISTS is an arbitrary SELECT statement, or *subquery*. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of EXISTS is TRUE; if the subquery returns no rows, the result of EXISTS is FALSE.

The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery.

The subquery will generally only be executed far enough to determine whether at least one row is returned, not all the way to completion. It is unwise to write a subquery that has any side-effects (such as calling sequence functions); whether the side-effects occur or not may be difficult to predict.

Since the result depends only on whether any rows are returned, and not on the contents of those rows, the output list of the subquery is normally uninteresting. A common coding convention is to write all EXISTS tests in the form EXISTS(SELECT 1 WHERE ...). There are exceptions to this rule however, such as subqueries that use INTERSECT.

This simple example is like an inner join on col2, but it produces at most one output row for each tab1 row, even if there are multiple matching tab2 rows:

```
SELECT col1 FROM tab1
    WHERE EXISTS(SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

IN (scalar form)

```
expression IN (value[, ...])
```

The right-hand side of this form of IN is a parenthesized list of scalar expressions. The result is TRUE if the left-hand expression's result is equal to any of the right-hand expressions. This is a shorthand notation for

```
expression = value1
OR
expression = value2
OR
...
```

Note that if the left-hand expression yields NULL, or if there are no equal right-hand values and at least one right-hand expression yields NULL, the result of the IN construct will be NULL, not FALSE. This is in accordance with SQL's normal rules for Boolean combinations of NULL values.

Note

This form of IN is not truly a subquery expression, but it seems best to document it in the same place as subquery IN.

IN (subquery form)

```
expression IN (subquery)
```

The right-hand side of this form of IN is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of IN is TRUE if any equal subquery row is found. The result is FALSE if no equal row is found (including the special case where the subquery returns no rows).

Note that if the left-hand expression yields NULL, or if there are no equal right-hand values and at least one right-hand row yields NULL, the result of the IN construct will be NULL, not FALSE. This is in accordance with SQL's normal rules for Boolean combinations of NULL values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

```
(expression, expression[, ...]) IN (subquery)
```

The right-hand side of this form of IN is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand list. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result. The result of IN is TRUE if any equal subquery row is found. The result is FALSE if no equal row is found (including the special case where the subquery returns no rows).

As usual, NULLs in the expressions or subquery rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (NULL). If all the row results are either unequal or NULL, with at least one NULL, then the result of IN is NULL.

NOT IN (scalar form)

```
expression NOT IN (value[, ...])
```

The right-hand side of this form of NOT IN is a parenthesized list of scalar expressions. The result is TRUE if the left-hand expression's result is unequal to all of the right-hand expressions. This is a shorthand notation for

```
expression <> value1
AND
expression <> value2
AND
...
```

Note that if the left-hand expression yields NULL, or if there are no equal right-hand values and at least one right-hand expression yields NULL, the result of the `NOT IN` construct will be NULL, not TRUE as one might naively expect. This is in accordance with SQL's normal rules for Boolean combinations of NULL values.

Tip

`x NOT IN y` is equivalent to `NOT (x IN y)` in all cases. However, NULLs are much more likely to trip up the novice when working with `NOT IN` than when working with `IN`. It's best to express your condition positively if possible.

NOT IN (subquery form)

```
expression NOT IN (subquery)
```

The right-hand side of this form of `NOT IN` is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of `NOT IN` is TRUE if only unequal subquery rows are found (including the special case where the subquery returns no rows). The result is FALSE if any equal row is found.

Note that if the left-hand expression yields NULL, or if there are no equal right-hand values and at least one right-hand row yields NULL, the result of the `NOT IN` construct will be NULL, not TRUE. This is in accordance with SQL's normal rules for Boolean combinations of NULL values.

As with `EXISTS`, it's unwise to assume that the subquery will be evaluated completely.

```
(expression, expression[, ...]) NOT IN (subquery)
```

The right-hand side of this form of `NOT IN` is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand list. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result. The result of `NOT IN` is TRUE if only unequal subquery rows are found (including the special case where the subquery returns no rows). The result is FALSE if any equal row is found.

As usual, NULLs in the expressions or subquery rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (NULL). If all the row results are either unequal or NULL, with at least one NULL, then the result of `NOT IN` is NULL.

ANY

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

The right-hand side of this form of `ANY` is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given *operator*, which must yield a Boolean result. The result of `ANY` is TRUE if any true result is obtained.

The result is FALSE if no true result is found (including the special case where the subquery returns no rows).

SOME is a synonym for ANY. IN is equivalent to = ANY.

Note that if there are no successes and at least one right-hand row yields NULL for the operator's result, the result of the ANY construct will be NULL, not FALSE. This is in accordance with SQL's normal rules for Boolean combinations of NULL values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

```
(expression, expression[, ...]) operator ANY (subquery)
(expression, expression[, ...]) operator SOME (subquery)
```

The right-hand side of this form of ANY is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand list. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given *operator*. Presently, only = and <> operators are allowed in row-wise ANY queries. The result of ANY is TRUE if any equal or unequal row is found, respectively. The result is FALSE if no such row is found (including the special case where the subquery returns no rows).

As usual, NULLs in the expressions or subquery rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (NULL). If there is at least one NULL row result, then the result of ANY cannot be FALSE; it will be TRUE or NULL.

ALL

```
expression operator ALL (subquery)
```

The right-hand side of this form of ALL is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given *operator*, which must yield a Boolean result. The result of ALL is TRUE if all rows yield TRUE (including the special case where the subquery returns no rows). The result is FALSE if any false result is found.

NOT IN is equivalent to <> ALL.

Note that if there are no failures but at least one right-hand row yields NULL for the operator's result, the result of the ALL construct will be NULL, not TRUE. This is in accordance with SQL's normal rules for Boolean combinations of NULL values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

```
(expression, expression[, ...]) operator ALL (subquery)
```

The right-hand side of this form of ALL is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand list. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given *operator*. Presently, only = and <> operators are allowed in row-wise ALL queries. The result of ALL is TRUE if all subquery rows are equal or unequal, respectively (including the special case where the subquery returns no rows). The result is FALSE if any row is found to be unequal or equal, respectively.

As usual, NULLs in the expressions or subquery rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the

rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (NULL). If there is at least one NULL row result, then the result of `ALL` cannot be TRUE; it will be FALSE or NULL.

Row-wise comparison

```
(expression, expression[, ...]) operator (subquery)
(expression, expression[, ...]) operator
 (expression, expression[, ...])
```

The left-hand side is a list of scalar expressions. The right-hand side can be either a list of scalar expressions of the same length, or a parenthesized subquery, which must return exactly as many columns as there are expressions on the left-hand side. Furthermore, the subquery cannot return more than one row. (If it returns zero rows, the result is taken to be NULL.) The left-hand side is evaluated and compared row-wise to the single subquery result row, or to the right-hand expression list. Presently, only = and <> operators are allowed in row-wise comparisons. The result is TRUE if the two rows are equal or unequal, respectively.

As usual, NULLs in the expressions or subquery rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of the row comparison is unknown (NULL).

Chapter 5. Type Conversion

5.1. Introduction

SQL queries can, intentionally or not, require mixing of different data types in the same expression. PostgreSQL has extensive facilities for evaluating mixed-type expressions.

In many cases a user will not need to understand the details of the type conversion mechanism. However, the implicit conversions done by PostgreSQL can affect the results of a query. When necessary, these results can be tailored by a user or programmer using *explicit* type coercion.

This chapter introduces the PostgreSQL type conversion mechanisms and conventions. Refer to the relevant sections in Chapter 3, *Data Types* and Chapter 4, *Functions and Operators* for more information on specific data types and allowed functions and operators.

The *Programmer's Guide* has more details on the exact algorithms used for implicit type conversion and coercion.

5.2. Overview

SQL is a strongly typed language. That is, every data item has an associated data type which determines its behavior and allowed usage. PostgreSQL has an extensible type system that is much more general and flexible than other RDBMS implementations. Hence, most type conversion behavior in PostgreSQL should be governed by general rules rather than by *ad hoc* heuristics, to allow mixed-type expressions to be meaningful even with user-defined types.

The PostgreSQL scanner/parser decodes lexical elements into only five fundamental categories: integers, floating-point numbers, strings, names, and key words. Most extended types are first tokenized into strings. The SQL language definition allows specifying type names with strings, and this mechanism can be used in PostgreSQL to start the parser down the correct path. For example, the query

```
tgl=> SELECT text 'Origin' AS "Label", point '(0,0)' AS "Value";
 Label  | Value
--------+-------
 Origin | (0,0)
(1 row)
```

has two literal constants, of type `text` and `point`. If a type is not specified for a string literal, then the placeholder type *unknown* is assigned initially, to be resolved in later stages as described below.

There are four fundamental SQL constructs requiring distinct type conversion rules in the PostgreSQL parser:

Operators

PostgreSQL allows expressions with prefix and postfix unary (one-argument) operators, as well as binary (two-argument) operators.

Function calls

Much of the PostgreSQL type system is built around a rich set of functions. Function calls have one or more arguments which, for any specific query, must be matched to the functions available in the system catalog. Since PostgreSQL permits function overloading, the function name alone does not

uniquely identify the function to be called; the parser must select the right function based on the data types of the supplied arguments.

Query targets

SQL `INSERT` and `UPDATE` statements place the results of expressions into a table. The expressions in the query must be matched up with, and perhaps converted to, the types of the target columns.

`UNION` and `CASE` constructs

Since all select results from a unionized `SELECT` statement must appear in a single set of columns, the types of the results of each `SELECT` clause must be matched up and converted to a uniform set. Similarly, the result expressions of a `CASE` construct must be coerced to a common type so that the `CASE` expression as a whole has a known output type.

Many of the general type conversion rules use simple conventions built on the PostgreSQL function and operator system tables. There are some heuristics included in the conversion rules to better support conventions for the SQL standard native types such as `smallint`, `integer`, and `real`.

The PostgreSQL parser uses the convention that all type conversion functions take a single argument of the source type and are named with the same name as the target type. Any function meeting these criteria is considered to be a valid conversion function, and may be used by the parser as such. This simple assumption gives the parser the power to explore type conversion possibilities without hardcoding, allowing extended user-defined types to use these same features transparently.

An additional heuristic is provided in the parser to allow better guesses at proper behavior for SQL standard types. There are several basic *type categories* defined: `boolean`, `numeric`, `string`, `bitstring`, `datetime`, `timespan`, `geometric`, `network`, and user-defined. Each category, with the exception of user-defined, has a *preferred type* which is preferentially selected when there is ambiguity. In the user-defined category, each type is its own preferred type. Ambiguous expressions (those with multiple candidate parsing solutions) can often be resolved when there are multiple possible built-in types, but they will raise an error when there are multiple choices for user-defined types.

All type conversion rules are designed with several principles in mind:

- Implicit conversions should never have surprising or unpredictable outcomes.
- User-defined types, of which the parser has no *a priori* knowledge, should be "higher" in the type hierarchy. In mixed-type expressions, native types shall always be converted to a user-defined type (of course, only if conversion is necessary).
- User-defined types are not related. Currently, PostgreSQL does not have information available to it on relationships between types, other than hardcoded heuristics for built-in types and implicit relationships based on available functions in the catalog.
- There should be no extra overhead from the parser or executor if a query does not need implicit type conversion. That is, if a query is well formulated and the types already match up, then the query should proceed without spending extra time in the parser and without introducing unnecessary implicit conversion functions into the query.

  Additionally, if a query usually requires an implicit conversion for a function, and if then the user defines an explicit function with the correct argument types, the parser should use this new function and will no longer do the implicit conversion using the old function.

5.3. Operators

The operand types of an operator invocation are resolved following the procedure below. Note that this procedure is indirectly affected by the precedence of the involved operators. See Section 1.4, "Lexical Precedence" for more information.

Operand Type Resolution

1. Check for an exact match in the `pg_operator` system catalog.

   - (Optional) If one argument of a binary operator is `unknown` type, then assume it is the same type as the other argument for this check. Other cases involving `unknown` will never find a match at this step.

2. Look for the best match.

   a. Make a list of all operators of the same name for which the input types match or can be coerced to match. (`unknown` literals are assumed to be coercible to anything for this purpose.) If there is only one, use it; else continue to the next step.

   b. Run through all candidates and keep those with the most exact matches on input types. Keep all candidates if none have any exact matches. If only one candidate remains, use it; else continue to the next step.

   c. Run through all candidates and keep those with the most exact or binary-compatible matches on input types. Keep all candidates if none have any exact or binary-compatible matches. If only one candidate remains, use it; else continue to the next step.

   d. Run through all candidates and keep those that accept preferred types at the most positions where type coercion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.

   e. If any input arguments are “unknown”, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the "string" category if any candidate accepts that category (this bias towards string is appropriate since an unknown-type literal does look like a string). Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Also note whether any of the candidates accept a preferred data type within the selected category. Now discard operator candidates that do not accept the selected type category; furthermore, if any candidate accepts a preferred type at a given argument position, discard candidates that accept non-preferred types for that argument.

   f. If only one candidate remains, use it. If no candidate or more than one candidate remains, then fail.

Examples

Example 5.1. Exponentiation Operator Type Resolution

There is only one exponentiation operator defined in the catalog, and it takes arguments of type `double precision`. The scanner assigns an initial type of `integer` to both arguments of this query expression:

```
tgl=> SELECT 2 ^ 3 AS "Exp";
 Exp
-----
   8
(1 row)
```

So the parser does a type conversion on both operands and the query is equivalent to

```
tgl=> SELECT CAST(2 AS double precision) ^ CAST(3 AS double precision)
 AS "Exp";
```

```
 Exp
-----
   8
(1 row)
```

or

```
tgl=> SELECT 2.0 ^ 3.0 AS "Exp";
 Exp
-----
   8
(1 row)
```

Note

This last form has the least overhead, since no functions are called to do implicit type conversion. This is not an issue for small queries, but may have an impact on the performance of queries involving large tables.

Example 5.2. String Concatenation Operator Type Resolution

A string-like syntax is used for working with string types as well as for working with complex extended types. Strings with unspecified type are matched with likely operator candidates.

An example with one unspecified argument:

```
tgl=> SELECT text 'abc' || 'def' AS "Text and Unknown";
 Text and Unknown
------------------
 abcdef
(1 row)
```

In this case the parser looks to see if there is an operator taking `text` for both arguments. Since there is, it assumes that the second argument should be interpreted as of type `text`.

Concatenation on unspecified types:

```
tgl=> SELECT 'abc' || 'def' AS "Unspecified";
 Unspecified
-------------
 abcdef
(1 row)
```

In this case there is no initial hint for which type to use, since no types are specified in the query. So, the parser looks for all candidate operators and finds that there are candidates accepting both string-category and bit-string-category inputs. Since string category is preferred when available, that category is selected, and then the “preferred type” for strings, `text`, is used as the specific type to resolve the unknown literals to.

Example 5.3. Absolute-Value and Factorial Operator Type Resolution

The PostgreSQL operator catalog has several entries for the prefix operator `@`, all of which implement absolute-value operations for various numeric data types. One of these entries is for type `float8`, which is the preferred type in the numeric category. Therefore, PostgreSQL will use that entry when faced with a non-numeric input:

```
tgl=> select @ text '-4.5' as "abs";
```

```
 abs
-----
 4.5
(1 row)
```

Here the system has performed an implicit text-to-float8 conversion before applying the chosen operator. We can verify that float8 and not some other type was used:

```
tgl=> select @ text '-4.5e500' as "abs";
ERROR:  Input '-4.5e500' is out of range for float8
```

On the other hand, the postfix operator `!` (factorial) is defined only for integer data types, not for float8. So, if we try a similar case with `!`, we get:

```
tgl=> select text '44' ! as "factorial";
ERROR:  Unable to identify a postfix operator '!' for type 'text'
        You may need to add parentheses or an explicit cast
```

This happens because the system can't decide which of the several possible `!` operators should be preferred. We can help it out with an explicit cast:

```
tgl=> select cast(text '44' as int8) ! as "factorial";
      factorial
---------------------
 2673996885588443136
(1 row)
```

5.4. Functions

The argument types of function calls are resolved according to the following steps.

Function Argument Type Resolution

1. Check for an exact match in the `pg_proc` system catalog. (Cases involving `unknown` will never find a match at this step.)

2. If no exact match appears in the catalog, see whether the function call appears to be a trivial type coercion request. This happens if the function call has just one argument and the function name is the same as the (internal) name of some data type. Furthermore, the function argument must be either an unknown-type literal or a type that is binary-compatible with the named data type. When these conditions are met, the function argument is coerced to the named data type without any explicit function call.

3. Look for the best match.

    a. Make a list of all functions of the same name with the same number of arguments for which the input types match or can be coerced to match. (`unknown` literals are assumed to be coercible to anything for this purpose.) If there is only one, use it; else continue to the next step.

    b. Run through all candidates and keep those with the most exact matches on input types. Keep all candidates if none have any exact matches. If only one candidate remains, use it; else continue to the next step.

    c. Run through all candidates and keep those with the most exact or binary-compatible matches on input types. Keep all candidates if none have any exact or binary-compatible matches. If only one candidate remains, use it; else continue to the next step.

d. Run through all candidates and keep those that accept preferred types at the most positions where type coercion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.

e. If any input arguments are `unknown`, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the `string` category if any candidate accepts that category (this bias towards string is appropriate since an unknown-type literal does look like a string). Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Also note whether any of the candidates accept a preferred data type within the selected category. Now discard candidates that do not accept the selected type category; furthermore, if any candidate accepts a preferred type at a given argument position, discard candidates that accept non-preferred types for that argument.

f. If only one candidate remains, use it. If no candidate or more than one candidate remains, then fail.

Examples

Example 5.4. Factorial Function Argument Type Resolution

There is only one `int4fac` function defined in the `pg_proc` catalog. So the following query automatically converts the `int2` argument to `int4`:

```
tgl=> SELECT int4fac(int2 '4');
 int4fac
---------
      24
(1 row)
```

and is actually transformed by the parser to

```
tgl=> SELECT int4fac(int4(int2 '4'));
 int4fac
---------
      24
(1 row)
```

Example 5.5. Substring Function Type Resolution

There are two `substr` functions declared in `pg_proc`. However, only one takes two arguments, of types `text` and `int4`.

If called with a string constant of unspecified type, the type is matched up directly with the only candidate function type:

```
tgl=> SELECT substr('1234', 3);
 substr
--------
     34
(1 row)
```

If the string is declared to be of type `varchar`, as might be the case if it comes from a table, then the parser will try to coerce it to become `text`:

```
tgl=> SELECT substr(varchar '1234', 3);
 substr
--------
     34
(1 row)
```

which is transformed by the parser to become

```
tgl=> SELECT substr(text(varchar '1234'), 3);
 substr
--------
     34
(1 row)
```

Note

Actually, the parser is aware that `text` and `varchar` are *binary-compatible*, meaning that one can be passed to a function that accepts the other without doing any physical conversion. Therefore, no explicit type conversion call is really inserted in this case.

And, if the function is called with an `int4`, the parser will try to convert that to `text`:

```
tgl=> SELECT substr(1234, 3);
 substr
--------
     34
(1 row)
```

which actually executes as

```
tgl=> SELECT substr(text(1234), 3);
 substr
--------
     34
(1 row)
```

This succeeds because there is a conversion function text(int4) in the system catalog.

5.5. Query Targets

Values to be inserted into a table are coerced to the destination column's data type according to the following steps.

Query Target Type Resolution

1. Check for an exact match with the target.

2. Otherwise, try to coerce the expression to the target type. This will succeed if the two types are known binary-compatible, or if there is a conversion function. If the expression is an unknown-type literal, the contents of the literal string will be fed to the input conversion routine for the target type.

3. If the target is a fixed-length type (e.g. `char` or `varchar` declared with a length) then try to find a sizing function for the target type. A sizing function is a function of the same name as the type, taking two arguments of which the first is that type and the second is an integer, and returning the same type. If one is found, it is applied, passing the column's declared length as the second parameter.

Example 5.6. `character` Storage Type Conversion

For a target column declared as `character(20)` the following query ensures that the target is sized correctly:

```
tgl=> CREATE TABLE vv (v character(20));
CREATE
tgl=> INSERT INTO vv SELECT 'abc' || 'def';
INSERT 392905 1
tgl=> SELECT v, length(v) FROM vv;
          v           | length
----------------------+--------
 abcdef               |     20
(1 row)
```

What has really happened here is that the two unknown literals are resolved to `text` by default, allowing the `||` operator to be resolved as `text` concatenation. Then the `text` result of the operator is coerced to `bpchar` ("blank-padded char", the internal name of the character data type) to match the target column type. (Since the parser knows that `text` and `bpchar` are binary-compatible, this coercion is implicit and does not insert any real function call.) Finally, the sizing function `bpchar(bpchar, integer)` is found in the system catalogs and applied to the operator's result and the stored column length. This type-specific function performs the required length check and addition of padding spaces.

5.6. `UNION` and `CASE` Constructs

SQL `UNION` constructs must match up possibly dissimilar types to become a single result set. The resolution algorithm is applied separately to each output column of a union query. The `INTERSECT` and `EXCEPT` constructs resolve dissimilar types in the same way as `UNION`. A `CASE` construct also uses the identical algorithm to match up its component expressions and select a result data type.

`UNION` and `CASE` Type Resolution

1. If all inputs are of type `unknown`, resolve as type `text` (the preferred type for string category). Otherwise, ignore the `unknown` inputs while choosing the type.
2. If the non-unknown inputs are not all of the same type category, fail.
3. If one or more non-unknown inputs are of a preferred type in that category, resolve as that type.
4. Otherwise, resolve as the type of the first non-unknown input.
5. Coerce all inputs to the selected type.

Examples

Example 5.7. Underspecified Types in a Union

```
tgl=> SELECT text 'a' AS "Text" UNION SELECT 'b';
 Text
------
 a
 b
(2 rows)
```

Here, the unknown-type literal `'b'` will be resolved as type text.

Example 5.8. Type Conversion in a Simple Union

```
tgl=> SELECT 1.2 AS "Double" UNION SELECT 1;
 Double
--------
      1
    1.2
(2 rows)
```

The literal `1.2` is of type `double precision`, the preferred type in the numeric category, so that type is used.

Example 5.9. Type Conversion in a Transposed Union

Here the output type of the union is forced to match the type of the first clause in the union:

```
tgl=> SELECT 1 AS "All integers"
tgl-> UNION SELECT CAST('2.2' AS REAL);
 All integers
--------------
            1
            2
(2 rows)
```

Since `REAL` is not a preferred type, the parser sees no reason to select it over `INTEGER` (which is what the 1 is), and instead falls back on the use-the-first-alternative rule. This example demonstrates that the preferred-type mechanism doesn't encode as much information as we'd like. Future versions of PostgreSQL may support a more general notion of type preferences.

Chapter 6. Arrays

PostgreSQL allows columns of a table to be defined as variable-length multidimensional arrays. Arrays of any built-in type or user-defined type can be created. To illustrate their use, we create this table:

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

As shown, an array data type is named by appending square brackets (`[]`) to the data type name of the array elements. The above query will create a table named `sal_emp` with columns including a `text` string (`name`), a one-dimensional array of type `integer` (`pay_by_quarter`), which represents the employee's salary by quarter, and a two-dimensional array of `text` (`schedule`), which represents the employee's weekly schedule.

Now we do some `INSERT`s. Observe that to write an array value, we enclose the element values within curly braces and separate them by commas. If you know C, this is not unlike the syntax for initializing structures. (More details appear below.)

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"talk", "consult"}, {"meeting"}}');
```

Now, we can run some queries on `sal_emp`. First, we show how to access a single element of an array at a time. This query retrieves the names of the employees whose pay changed in the second quarter:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

 name
-------
 Carol
(1 row)
```

The array subscript numbers are written within square brackets. By default PostgreSQL uses the "one-based" numbering convention for arrays, that is, an array of *n* elements starts with `array[1]` and ends with `array[`*n*`]`.

This query retrieves the third quarter pay of all employees:

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

We can also access arbitrary rectangular slices of an array, or subarrays. An array slice is denoted by writing *`lower subscript : upper subscript`* for one or more array dimensions. This query retrieves the first item on Bill's schedule for the first two days of the week:

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

      schedule
--------------------
 {{meeting},{""}}
(1 row)
```

We could also have written

```
SELECT schedule[1:2][1] FROM sal_emp WHERE name = 'Bill';
```

with the same result. An array subscripting operation is taken to represent an array slice if any of the subscripts are written in the form *`lower : upper`*. A lower bound of 1 is assumed for any subscript where only one value is specified.

An array value can be replaced completely:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

or updated at a single element:

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

or updated in a slice:

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

An array can be enlarged by assigning to an element adjacent to those already present, or by assigning to a slice that is adjacent to or overlaps the data already present. For example, if an array value currently has 4 elements, it will have five elements after an update that assigns to `array[5]`. Currently, enlargement in this fashion is only allowed for one-dimensional arrays, not multidimensional arrays.

Array slice assignment allows creation of arrays that do not use one-based subscripts. For example one might assign to `array[-2:7]` to create an array with subscript values running from -2 to 7.

The syntax for `CREATE TABLE` allows fixed-length arrays to be defined:

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

However, the current implementation does not enforce the array size limits --- the behavior is the same as for arrays of unspecified length.

Actually, the current implementation does not enforce the declared number of dimensions either. Arrays of a particular element type are all considered to be of the same type, regardless of size or number of dimensions. So, declaring number of dimensions or sizes in `CREATE TABLE` is simply documentation, it does not affect runtime behavior.

The current dimensions of any array value can be retrieved with the `array_dims` function:

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

 array_dims
------------
 [1:2][1:1]
(1 row)
```

`array_dims` produces a `text` result, which is convenient for people to read but perhaps not so convenient for programs.

To search for a value in an array, you must check each value of the array. This can be done by hand (if you know the size of the array):

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

However, this quickly becomes tedious for large arrays, and is not helpful if the size of the array is unknown. Although it is not part of the primary PostgreSQL distribution, there is an extension available that defines new functions and operators for iterating over array values. Using this, the above query could be:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1:4] *= 10000;
```

To search the entire array (not just specified columns), you could use:

```
SELECT * FROM sal_emp WHERE pay_by_quarter *= 10000;
```

In addition, you could find rows where the array had all values equal to 10 000 with:

```
SELECT * FROM sal_emp WHERE pay_by_quarter **= 10000;
```

To install this optional module, look in the `contrib/array` directory of the PostgreSQL source distribution.

Tip

Arrays are not sets; using arrays in the manner described in the previous paragraph is often a sign of database misdesign. The array field should generally be split off into a separate table. Tables can obviously be searched easily.

Note

A limitation of the present array implementation is that individual elements of an array cannot be SQL NULLs. The entire array can be set to NULL, but you can't have an array with some elements NULL and some not. Fixing this is on the to-do list.

Array input and output syntax. The external representation of an array value consists of items that are interpreted according to the I/O conversion rules for the array's element type, plus decoration that indicates the array structure. The decoration consists of curly braces (`{` and `}`) around the array value plus delimiter characters between adjacent items. The delimiter character is usually a comma (`,`) but can be something else: it is determined by the `typdelim` setting for the array's element type. (Among the standard datatypes provided in the PostgreSQL distribution, type `box` uses a semicolon (`;`) but all the others use comma.) In a multidimensional array, each dimension (row, plane, cube, etc.) gets its own level of curly braces, and delimiters must be written between adjacent curly-braced entities of the same level. You may write whitespace before a left brace, after a right brace, or before any individual item string. Whitespace after

an item is not ignored, however: after skipping leading whitespace, everything up to the next right brace or delimiter is taken as the item value.

Quoting array elements. As shown above, when writing an array value you may write double quotes around any individual array element. You *must* do so if the element value would otherwise confuse the array-value parser. For example, elements containing curly braces, commas (or whatever the delimiter character is), double quotes, backslashes, or leading white space must be double-quoted. To put a double quote or backslash in an array element value, precede it with a backslash. Alternatively, you can use backslash-escaping to protect all data characters that would otherwise be taken as array syntax or ignorable white space.

The array output routine will put double quotes around element values if they are empty strings or contain curly braces, delimiter characters, double quotes, backslashes, or white space. Double quotes and backslashes embedded in element values will be backslash-escaped. For numeric datatypes it is safe to assume that double quotes will never appear, but for textual datatypes one should be prepared to cope with either presence or absence of quotes. (This is a change in behavior from pre-7.2 PostgreSQL releases.)

Tip

Remember that what you write in an SQL query will first be interpreted as a string literal, and then as an array. This doubles the number of backslashes you need. For example, to insert a `text` array value containing a backslash and a double quote, you'd need to write

```
INSERT ... VALUES ('{"\\\\","\\""}');
```

The string-literal processor removes one level of backslashes, so that what arrives at the array-value parser looks like `{"\\","\""}`. In turn, the strings fed to the `text` data type's input routine become `\` and `"` respectively. (If we were working with a data type whose input routine also treated backslashes specially, `bytea` for example, we might need as many as eight backslashes in the query to get one backslash into the stored array element.)

Chapter 7. Indexes

Indexes are a common way to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index. But indexes also add overhead to the database system as a whole, so they should be used sensibly.

7.1. Introduction

The classical example for the need of an index is if there is a table similar to this:

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

and the application requires a lot of queries of the form

```
SELECT content FROM test1 WHERE id = constant;
```

Ordinarily, the system would have to scan the entire `test1` table row by row to find all matching entries. If there are a lot of rows in `test1` and only a few rows (possibly zero or one) returned by the query, then this is clearly an inefficient method. If the system were instructed to maintain an index on the `id` column, then it could use a more efficient method for locating matching rows. For instance, it might only have to walk a few levels deep into a search tree.

A similar approach is used in most books of non-fiction: Terms and concepts that are frequently looked up by readers are collected in an alphabetic index at the end of the book. The interested reader can scan the index relatively quickly and flip to the appropriate page, and would not have to read the entire book to find the interesting location. As it is the task of the author to anticipate the items that the readers are most likely to look up, it is the task of the database programmer to foresee which indexes would be of advantage.

The following command would be used to create the index on the `id` column, as discussed:

```
CREATE INDEX test1_id_index ON test1 (id);
```

The name `test1_id_index` can be chosen freely, but you should pick something that enables you to remember later what the index was for.

To remove an index, use the `DROP INDEX` command. Indexes can be added to and removed from tables at any time.

Once the index is created, no further intervention is required: the system will use the index when it thinks it would be more efficient than a sequential table scan. But you may have to run the `ANALYZE` command regularly to update statistics to allow the query planner to make educated decisions. Also read Chapter 11, *Performance Tips* for information about how to find out whether an index is used and when and why the planner may choose to *not* use an index.

Indexes can benefit `UPDATE`s and `DELETE`s with search conditions. Indexes can also be used in join queries. Thus, an index defined on a column that is part of a join condition can significantly speed up queries with joins.

When an index is created, the system has to keep it synchronized with the table. This adds overhead to data manipulation operations. Therefore indexes that are non-essential or do not get used at all should be removed. Note that a query or data manipulation command can use at most one index per table.

7.2. Index Types

PostgreSQL provides several index types: B-tree, R-tree, GiST, and Hash. Each index type is more appropriate for a particular query type because of the algorithm it uses. By default, the `CREATE INDEX` command will create a B-tree index, which fits the most common situations. In particular, the PostgreSQL query optimizer will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators: <, <=, =, >=, >

R-tree indexes are especially suited for spatial data. To create an R-tree index, use a command of the form

```
CREATE INDEX name ON table USING RTREE (column);
```

The PostgreSQL query optimizer will consider using an R-tree index whenever an indexed column is involved in a comparison using one of these operators: <<, &<, &>, >>, @, ~=, && (Refer to Section 4.9, “Geometric Functions and Operators” about the meaning of these operators.)

The query optimizer will consider using a hash index whenever an indexed column is involved in a comparison using the = operator. The following command is used to create a hash index:

```
CREATE INDEX name ON table USING HASH (column);
```

Note

Because of the limited utility of hash indexes, a B-tree index should generally be preferred over a hash index. We do not have sufficient evidence that hash indexes are actually faster than B-trees even for = comparisons. Moreover, hash indexes require coarser locks; see Section 9.7, “Locking and Indexes”.

The B-tree index is an implementation of Lehman-Yao high-concurrency B-trees. The R-tree index method implements standard R-trees using Guttman's quadratic split algorithm. The hash index is an implementation of Litwin's linear hashing. We mention the algorithms used solely to indicate that all of these access methods are fully dynamic and do not have to be optimized periodically (as is the case with, for example, static hash access methods).

7.3. Multicolumn Indexes

An index can be defined on more than one column. For example, if you have a table of this form:

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
);
```

(Say, you keep your `/dev` directory in a database...) and you frequently make queries like

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

then it may be appropriate to define an index on the columns `major` and `minor` together, e.g.,

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

Currently, only the B-tree and GiST implementations support multicolumn indexes. Up to 16 columns may be specified. (This limit can be altered when building PostgreSQL; see the file `pg_config.h`.)

The query optimizer can use a multicolumn index for queries that involve the first *n* consecutive columns in the index (when used with appropriate operators), up to the total number of columns specified in the index definition. For example, an index on `(a, b, c)` can be used in queries involving all of `a`, `b`, and `c`, or in queries involving both `a` and `b`, or in queries involving only `a`, but not in other combinations. (In a query involving `a` and `c` the optimizer might choose to use the index for `a` only and treat `c` like an ordinary unindexed column.)

Multicolumn indexes can only be used if the clauses involving the indexed columns are joined with `AND`. For instance,

```
SELECT name FROM test2 WHERE major = constant OR minor = constant;
```

cannot make use of the index `test2_mm_idx` defined above to look up both columns. (It can be used to look up only the `major` column, however.)

Multicolumn indexes should be used sparingly. Most of the time, an index on a single column is sufficient and saves space and time. Indexes with more than three columns are almost certainly inappropriate.

7.4. Unique Indexes

Indexes may also be used to enforce uniqueness of a column's value, or the uniqueness of the combined values of more than one column.

```
CREATE UNIQUE INDEX name ON table (column [, ...]);
```

Currently, only B-tree indexes can be declared unique.

When an index is declared unique, multiple table rows with equal indexed values will not be allowed. NULL values are not considered equal.

PostgreSQL automatically creates unique indexes when a table is declared with a unique constraint or a primary key, on the columns that make up the primary key or unique columns (a multicolumn index, if appropriate), to enforce that constraint. A unique index can be added to a table at any later time, to add a unique constraint.

Note

The preferred way to add a unique constraint to a table is `ALTER TABLE ... ADD CONSTRAINT`. The use of indexes to enforce unique constraints could be considered an implementation detail that should not be accessed directly.

7.5. Functional Indexes

For a *functional index*, an index is defined on the result of a function applied to one or more columns of a single table. Functional indexes can be used to obtain fast access to data based on the result of function calls.

For example, a common way to do case-insensitive comparisons is to use the `lower` function:

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

This query can use an index, if one has been defined on the result of the `lower(column)` operation:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

The function in the index definition can take more than one argument, but they must be table columns, not constants. Functional indexes are always single-column (namely, the function result) even if the function uses more than one input field; there cannot be multicolumn indexes that contain function calls.

Tip

The restrictions mentioned in the previous paragraph can easily be worked around by defining a custom function to use in the index definition that computes any desired result internally.

7.6. Operator Classes

An index definition may specify an *operator class* for each column of an index.

```
CREATE INDEX name ON table (column opclass [, ...]);
```

The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on four-byte integers would use the `int4_ops` class; this operator class includes comparison functions for four-byte integers. In practice the default operator class for the column's data type is usually sufficient. The main point of having operator classes is that for some data types, there could be more than one meaningful ordering. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index. There are also some operator classes with special purposes:

- The operator classes `box_ops` and `bigbox_ops` both support R-tree indexes on the `box` data type. The difference between them is that `bigbox_ops` scales box coordinates down, to avoid floating-point exceptions from doing multiplication, addition, and subtraction on very large floating-point coordinates. If the field on which your rectangles lie is about 20 000 units square or larger, you should use `bigbox_ops`.

The following query shows all defined operator classes:

```
SELECT am.amname AS acc_method,
       opc.opcname AS ops_name,
       opr.oprname AS ops_comp
    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 acc_method, ops_name, ops_comp
```

7.7. Keys

Author

Written by Herouth Maoz (<herouth@oumail.openu.ac.il>). This originally appeared on the User's Mailing List on 1998-03-02 in response to the question: "What is the difference between PRIMARY KEY and UNIQUE constraints?".

Subject: Re: [QUESTIONS] PRIMARY KEY | UNIQUE

What's the difference between:

PRIMARY KEY(fields,...) and
UNIQUE (fields,...)

- Is this an alias?
- If PRIMARY KEY is already unique, then why
  is there another kind of key named UNIQUE?

A primary key is the field(s) used to identify a specific row. For example, Social Security numbers identifying a person.

A simply UNIQUE combination of fields has nothing to do with identifying the row. It's simply an integrity constraint. For example, I have collections of links. Each collection is identified by a unique number, which is the primary key. This key is used in relations.

However, my application requires that each collection will also have a unique name. Why? So that a human being who wants to modify a collection will be able to identify it. It's much harder to know, if you have two collections named “Life Science”, the one tagged 24433 is the one you need, and the one tagged 29882 is not.

So, the user selects the collection by its name. We therefore make sure, within the database, that names are unique. However, no other table in the database relates to the collections table by the collection Name. That would be very inefficient.

Moreover, despite being unique, the collection name does not actually define the collection! For example, if somebody decided to change the name of the collection from “Life Science” to “Biology”, it will still be the same collection, only with a different name. As long as the name is unique, that's OK.

So:

- Primary key:
  - Is used for identifying the row and relating to it.
  - Is impossible (or hard) to update.
  - Should not allow NULLs.

- Unique field(s):
  - Are used as an alternative access to the row.
  - Are updatable, so long as they are kept unique.
  - NULLs are acceptable.

As for why no non-unique keys are defined explicitly in standard SQL syntax? Well, you must understand that indexes are implementation-dependent. SQL does not define the implementation, merely the relations between data in the database. PostgreSQL does allow non-unique indexes, but indexes used to enforce SQL keys are always unique.

Thus, you may query a table by any combination of its columns, despite the fact that you don't have an index on these columns. The indexes are merely an implementation aid that each RDBMS offers you, in order to cause commonly used queries to be done more efficiently. Some RDBMS may give you additional measures, such as keeping a key stored in main memory. They will have a special command, for example

```
CREATE MEMSTORE ON table COLUMNS cols
```

(This is not an existing command, just an example.)

In fact, when you create a primary key or a unique combination of fields, nowhere in the SQL specification does it say that an index is created, nor that the retrieval of data by the key is going to be more efficient than a sequential scan!

So, if you want to use a combination of fields that is not unique as a secondary key, you really don't have to specify anything - just start retrieving by that combination! However, if you want to make the retrieval efficient, you'll have to resort to the means your RDBMS provider gives you - be it an index, my imaginary `MEMSTORE` command, or an intelligent RDBMS that creates indexes without your knowledge based on the fact that you have sent it many queries based on a specific combination of keys... (It learns from experience).

7.8. Partial Indexes

A *partial index* is an index built over a subset of a table; the subset is defined by a conditional expression (called the *predicate* of the partial index). The index contains entries for only those table rows that satisfy the predicate.

A major motivation for partial indexes is to avoid indexing common values. Since a query searching for a common value (one that accounts for more than a few percent of all the table rows) will not use the index anyway, there is no point in keeping those rows in the index at all. This reduces the size of the index, which will speed up queries that do use the index. It will also speed up many table update operations because the index does not need to be updated in all cases. Example 7.1, “Setting up a Partial Index to Exclude Common Values” shows a possible application of this idea.

Example 7.1. Setting up a Partial Index to Exclude Common Values

Suppose you are storing web server access logs in a database. Most accesses originate from the IP range of your organization but some are from elsewhere (say, employees on dial-up connections). If your searches by IP are primarily for outside accesses, you probably do not need to index the IP range that corresponds to your organization's subnet.

Assume a table like this:

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

To create a partial index that suits our example, use a command such as this:

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
    WHERE NOT (client_ip > inet '192.168.100.0' AND client_ip < inet
 '192.168.100.255');
```

A typical query that can use this index would be:

```
SELECT * FROM access_log WHERE url = '/index.html' AND client_ip =
 inet '212.78.10.32';
```

A query that cannot use this index is:

```
SELECT * FROM access_log WHERE client_ip = inet '192.168.100.23';
```

Observe that this kind of partial index requires that the common values be predetermined. If the distribution of values is inherent (due to the nature of the application) and static (not changing over time), this is not difficult, but if the common values are merely due to the coincidental data load this can require a lot of maintenance work.

Another possibility is to exclude values from the index that the typical query workload is not interested in; this is shown in Example 7.2, “Setting up a Partial Index to Exclude Uninteresting Values”. This results in the same advantages as listed above, but it prevents the “uninteresting” values from being accessed via that index at all, even if an index scan might be profitable in that case. Obviously, setting up partial indexes for this kind of scenario will require a lot of care and experimentation.

Example 7.2. Setting up a Partial Index to Exclude Uninteresting Values

If you have a table that contains both billed and unbilled orders, where the unbilled orders take up a small fraction of the total table and yet those are the most-accessed rows, you can improve performance by creating an index on just the unbilled rows. The command to create the index would look like this:

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
    WHERE billed is not true;
```

A possible query to use this index would be

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

However, the index can also be used in queries that do not involve `order_nr` at all, e.g.,

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

This is not as efficient as a partial index on the `amount` column would be, since the system has to scan the entire index. Yet, if there are relatively few unbilled orders, using this partial index just to find the unbilled orders could be a win.

Note that this query cannot use this index:

```
SELECT * FROM orders WHERE order_nr = 3501;
```

The order 3501 may be among the billed or among the unbilled orders.

Example 7.2, “Setting up a Partial Index to Exclude Uninteresting Values” also illustrates that the indexed column and the column used in the predicate do not need to match. PostgreSQL supports partial indexes with arbitrary predicates, so long as only columns of the table being indexed are involved. However, keep in mind that the predicate must match the conditions used in the queries that are supposed to benefit from the index. To be precise, a partial index can be used in a query only if the system can recognize that the query's WHERE condition mathematically *implies* the index's predicate. PostgreSQL does not have a sophisticated theorem prover that can recognize mathematically equivalent predicates that are written in different forms. (Not only is such a general theorem prover extremely difficult to create, it would probably be too slow to be of any real use.) The system can recognize simple inequality implications, for example “x < 1” implies “x < 2”; otherwise the predicate condition must exactly match the query's WHERE condition or the index will not be recognized to be usable.

A third possible use for partial indexes does not require the index to be used in queries at all. The idea here is to create a unique index over a subset of a table, as in Example 7.3, “Setting up a Partial Unique Index”. This enforces uniqueness among the rows that satisfy the index predicate, without constraining those that do not.

Example 7.3. Setting up a Partial Unique Index

Suppose that we have a table describing test outcomes. We wish to ensure that there is only one “successful” entry for a given subject and target combination, but there might be any number of “unsuccessful” entries. Here is one way to do it:

```
CREATE TABLE tests (subject text,
                    target text,
                    success bool,
                    ...);
CREATE UNIQUE INDEX tests_success_constraint ON tests (subject,
 target)
    WHERE success;
```

This is a particularly efficient way of doing it when there are few successful trials and many unsuccessful ones.

Finally, a partial index can also be used to override the system's query plan choices. It may occur that data sets with peculiar distributions will cause the system to use an index when it really should not. In that case the index can be set up so that it is not available for the offending query. Normally, PostgreSQL makes reasonable choices about index usage (e.g., it avoids them when retrieving common values, so the earlier example really only saves index size, it is not required to avoid index usage), and grossly incorrect plan choices are cause for a bug report.

Keep in mind that setting up a partial index indicates that you know at least as much as the query planner knows, in particular you know when an index might be profitable. Forming this knowledge requires experience and understanding of how indexes in PostgreSQL work. In most cases, the advantage of a partial index over a regular index will not be much.

More information about partial indexes can be found in [ston89b], [olson93], and [seshadri95].

7.9. Examining Index Usage

Although indexes in PostgreSQL do not need maintenance and tuning, it is still important to check which indexes are actually used by the real-life query workload. Examining index usage is done with the `EXPLAIN` command; its application for this purpose is illustrated in Section 11.1, “Using `EXPLAIN`”.

It is difficult to formulate a general procedure for determining which indexes to set up. There are a number of typical cases that have been shown in the examples throughout the previous sections. A good deal of experimentation will be necessary in most cases. The rest of this section gives some tips for that.

- Always run `ANALYZE` first. This command collects statistics about the distribution of the values in the table. This information is required to guess the number of rows returned by a query, which is needed by the planner to assign realistic costs to each possible query plan. In absence of any real statistics, some default values are assumed, which are almost certain to be inaccurate. Examining an application's index usage without having run `ANALYZE` is therefore a lost cause.

- Use real data for experimentation. Using test data for setting up indexes will tell you what indexes you need for the test data, but that is all.

  It is especially fatal to use proportionally reduced data sets. While selecting 1000 out of 100000 rows could be a candidate for an index, selecting 1 out of 100 rows will hardly be, because the 100 rows will probably fit within a single disk page, and there is no plan that can beat sequentially fetching 1 disk page.

  Also be careful when making up test data, which is often unavoidable when the application is not in production use yet. Values that are very similar, completely random, or inserted in sorted order will skew the statistics away from the distribution that real data would have.

- When indexes are not used, it can be useful for testing to force their use. There are run-time parameters that can turn off various plan types (described in the *Administrator's Guide*). For instance, turning off sequential scans (`enable_seqscan`) and nested-loop joins (`enable_nestloop`), which are the

most basic plans, will force the system to use a different plan. If the system still chooses a sequential scan or nested-loop join then there is probably a more fundamental problem for why the index is not used, for example, the query condition does not match the index. (What kind of query can use what kind of index is explained in the previous sections.)

- If forcing index usage does use the index, then there are two possibilities: Either the system is right and using the index is indeed not appropriate, or the cost estimates of the query plans are not reflecting reality. So you should time your query with and without indexes. The `EXPLAIN ANALYZE` command can be useful here.

- If it turns out that the cost estimates are wrong, there are, again, two possibilities. The total cost is computed from the per-row costs of each plan node times the selectivity estimate of the plan node. The costs of the plan nodes can be tuned with run-time parameters (described in the *Administrator's Guide*). An inaccurate selectivity estimate is due to insufficient statistics. It may be possible to help this by tuning the statistics-gathering parameters (see `ALTER TABLE` reference).

  If you do not succeed in adjusting the costs to be more appropriate, then you may have to resort to forcing index usage explicitly. You may also want to contact the PostgreSQL developers to examine the issue.

Chapter 8. Inheritance

Let's create two tables. The capitals table contains state capitals which are also cities. Naturally, the capitals table should inherit from cities.

```
CREATE TABLE cities (
    name            text,
    population      float,
    altitude        int     -- (in ft)
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

In this case, a row of capitals *inherits* all attributes (name, population, and altitude) from its parent, cities. The type of the attribute name is `text`, a native PostgreSQL type for variable length ASCII strings. The type of the attribute population is `float`, a native PostgreSQL type for double precision floating-point numbers. State capitals have an extra attribute, state, that shows their state. In PostgreSQL, a table can inherit from zero or more other tables, and a query can reference either all rows of a table or all rows of a table plus all of its descendants.

Note

The inheritance hierarchy is actually a directed acyclic graph.

For example, the following query finds the names of all cities, including state capitals, that are located at an altitude over 500ft:

```
SELECT name, altitude
    FROM cities
    WHERE altitude > 500;
```

which returns:

```
+----------+----------+
|name      | altitude |
+----------+----------+
|Las Vegas | 2174     |
+----------+----------+
|Mariposa  | 1953     |
+----------+----------+
|Madison   | 845      |
+----------+----------+
```

On the other hand, the following query finds all the cities that are not state capitals and are situated at an altitude of 500ft or higher:

```
SELECT name, altitude
    FROM ONLY cities
    WHERE altitude > 500;
```

```
+----------+----------+
|name      | altitude |
+----------+----------+
|Las Vegas | 2174     |
+----------+----------+
|Mariposa  | 1953     |
+----------+----------+
```

Here the "ONLY" before cities indicates that the query should be run over only cities and not tables below cities in the inheritance hierarchy. Many of the commands that we have already discussed -- `SELECT`, `UPDATE` and `DELETE` -- support this "ONLY" notation.

In some cases you may wish to know which table a particular tuple originated from. There is a system column called `TABLEOID` in each table which can tell you the originating table:

```
SELECT c.tableoid, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
```

which returns:

```
+---------+----------+----------+
|tableoid |name      | altitude |
+---------+----------+----------+
|37292    |Las Vegas | 2174     |
+---------+----------+----------+
|37280    |Mariposa  | 1953     |
+---------+----------+----------+
|37280    |Madison   | 845      |
+---------+----------+----------+
```

If you do a join with pg_class you can see the actual table name:

```
SELECT p.relname, c.name, c.altitude
FROM cities c, pg_class p
WHERE c.altitude > 500 and c.tableoid = p.oid;
```

which returns:

```
+---------+----------+----------+
|relname  |name      | altitude |
+---------+----------+----------+
|capitals |Las Vegas | 2174     |
+---------+----------+----------+
|cities   |Mariposa  | 1953     |
+---------+----------+----------+
|cities   |Madison   | 845      |
+---------+----------+----------+
```

Deprecated

In previous versions of PostgreSQL, the default was not to get access to child tables. This was found to be error prone and is also in violation of SQL99. Under the old syntax, to get the sub-tables you append `*` to the table name. For example

```
SELECT * from cities*;
```

You can still explicitly specify scanning child tables by appending `*`, as well as explicitly specify not scanning child tables by writing "ONLY". But beginning in version 7.1, the default behavior for an undecorated table name is to scan its child tables too, whereas before the default was not to do so. To get the old default behavior, set the configuration option `SQL_Inheritance` to off, e.g.,

```
SET SQL_Inheritance TO OFF;
```

or add a line in your `postgresql.conf` file.

A limitation of the inheritance feature is that indexes (including unique constraints) and foreign key constraints only apply to single tables, not to their inheritance children. Thus, in the above example, specifying that another table's column `REFERENCES cities(name)` would allow the other table to contain city names but not capital names. This deficiency will probably be fixed in some future release.

Chapter 9. Multiversion Concurrency Control

Multiversion Concurrency Control (MVCC) is an advanced technique for improving database performance in a multiuser environment. Vadim Mikheev (<vadim@krs.ru>) provided the implementation for PostgreSQL.

9.1. Introduction

Unlike most other database systems which use locks for concurrency control, PostgreSQL maintains data consistency by using a multiversion model. This means that while querying a database each transaction sees a snapshot of data (a *database version*) as it was some time ago, regardless of the current state of the underlying data. This protects the transaction from viewing inconsistent data that could be caused by (other) concurrent transaction updates on the same data rows, providing *transaction isolation* for each database session.

The main difference between multiversion and lock models is that in MVCC locks acquired for querying (reading) data don't conflict with locks acquired for writing data and so reading never blocks writing and writing never blocks reading.

9.2. Transaction Isolation

The ANSI/ISO SQL standard defines four levels of transaction isolation in terms of three phenomena that must be prevented between concurrent transactions. These undesirable phenomena are:

dirty reads

A transaction reads data written by concurrent uncommitted transaction.

non-repeatable reads

A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).

phantom read

A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

The four transaction isolation levels and the corresponding behaviors are described in Table 9.1, “Isolation Levels”.

Table 9.1. SQL Transaction Isolation Levels

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

PostgreSQL offers the read committed and serializable isolation levels.

9.3. Read Committed Isolation Level

Read Committed is the default isolation level in PostgreSQL. When a transaction runs on this isolation level, a `SELECT` query sees only data committed before the query began and never sees either uncommitted data or changes committed during query execution by concurrent transactions. (However, the `SELECT` does see the effects of previous updates executed within this same transaction, even though they are not yet committed.) Notice that two successive `SELECT`s can see different data, even though they are within a single transaction, when other transactions commit changes during execution of the first `SELECT`.

If a target row found by a query while executing an `UPDATE` statement (or `DELETE` or `SELECT FOR UPDATE`) has already been updated by a concurrent uncommitted transaction then the second transaction that tries to update this row will wait for the other transaction to commit or rollback. In the case of rollback, the waiting transaction can proceed to change the row. In the case of commit (and if the row still exists; i.e. was not deleted by the other transaction), the query will be re-executed for this row to check that the new row version still satisfies the query search condition. If the new row version satisfies the query search condition then the row will be updated (or deleted or marked for update). Note that the starting point for the update will be the new row version; moreover, after the update the doubly-updated row is visible to subsequent `SELECT`s in the current transaction. Thus, the current transaction is able to see the effects of the other transaction for this specific row.

The partial transaction isolation provided by Read Committed level is adequate for many applications, and this level is fast and simple to use. However, for applications that do complex queries and updates, it may be necessary to guarantee a more rigorously consistent view of the database than the Read Committed level provides.

9.4. Serializable Isolation Level

Serializable provides the highest transaction isolation. This level emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. However, applications using this level must be prepared to retry transactions due to serialization failures.

When a transaction is on the serializable level, a `SELECT` query sees only data committed before the transaction began and never sees either uncommitted data or changes committed during transaction execution by concurrent transactions. (However, the `SELECT` does see the effects of previous updates executed within this same transaction, even though they are not yet committed.) This is different from Read Committed in that the `SELECT` sees a snapshot as of the start of the transaction, not as of the start of the current query within the transaction.

If a target row found by a query while executing an `UPDATE` statement (or `DELETE` or `SELECT FOR UPDATE`) has already been updated by a concurrent uncommitted transaction then the second transaction that tries to update this row will wait for the other transaction to commit or rollback. In the case of rollback, the waiting transaction can proceed to change the row. In the case of a concurrent transaction commit, a serializable transaction will be rolled back with the message

```
ERROR:  Can't serialize access due to concurrent update
```

because a serializable transaction cannot modify rows changed by other transactions after the serializable transaction began.

When the application receives this error message, it should abort the current transaction and then retry the whole transaction from the beginning. The second time through, the transaction sees the previously-committed change as part of its initial view of the database, so there is no logical conflict in using the new version of the row as the starting point for the new transaction's update. Note that only updating transactions may need to be retried --- read-only transactions never have serialization conflicts.

The Serializable transaction level provides a rigorous guarantee that each transaction sees a wholly consistent view of the database. However, the application has to be prepared to retry transactions when concurrent updates make it impossible to sustain the illusion of serial execution, and the cost of redoing complex transactions may be significant. So this level is recommended only when update queries contain logic sufficiently complex that they may give wrong answers in the Read Committed level.

9.5. Data consistency checks at the application level

Because readers in PostgreSQL don't lock data, regardless of transaction isolation level, data read by one transaction can be overwritten by another concurrent transaction. In other words, if a row is returned by `SELECT` it doesn't mean that the row still exists at the time it is returned (i.e., sometime after the current transaction began); the row might have been modified or deleted by an already-committed transaction that committed after this one started. Even if the row is still valid "now", it could be changed or deleted before the current transaction does a commit or rollback.

Another way to think about it is that each transaction sees a snapshot of the database contents, and concurrently executing transactions may very well see different snapshots. So the whole concept of "now" is somewhat suspect anyway. This is not normally a big problem if the client applications are isolated from each other, but if the clients can communicate via channels outside the database then serious confusion may ensue.

To ensure the current existence of a row and protect it against concurrent updates one must use `SELECT FOR UPDATE` or an appropriate `LOCK TABLE` statement. (`SELECT FOR UPDATE` locks just the returned rows against concurrent updates, while `LOCK TABLE` protects the whole table.) This should be taken into account when porting applications to PostgreSQL from other environments.

Note

Before version 6.5 PostgreSQL used read-locks and so the above consideration is also the case when upgrading to 6.5 (or higher) from previous PostgreSQL versions.

9.6. Locking and Tables

PostgreSQL provides various lock modes to control concurrent access to data in tables. Some of these lock modes are acquired by PostgreSQL automatically before statement execution, while others are provided to be used by applications. All lock modes acquired in a transaction are held for the duration of the transaction.

9.6.1. Table-level locks

AccessShareLock

A read-lock mode acquired automatically on tables being queried.

Conflicts with AccessExclusiveLock only.

RowShareLock

Acquired by `SELECT FOR UPDATE` and `LOCK TABLE` for `IN ROW SHARE MODE` statements.

Conflicts with ExclusiveLock and AccessExclusiveLock modes.

RowExclusiveLock

Acquired by `UPDATE`, `DELETE`, `INSERT` and `LOCK TABLE` for `IN ROW EXCLUSIVE MODE` statements.

Conflicts with ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

ShareUpdateExclusiveLock

Acquired by `VACUUM` (without `FULL`) and `LOCK TABLE` table for `IN SHARE UPDATE EXCLUSIVE MODE` statements.

Conflicts with ShareUpdateExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

ShareLock

Acquired by `CREATE INDEX` and `LOCK TABLE` table for `IN SHARE MODE` statements.

Conflicts with RowExclusiveLock, ShareUpdateExclusiveLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

ShareRowExclusiveLock

Acquired by `LOCK TABLE` for `IN SHARE ROW EXCLUSIVE MODE` statements.

Conflicts with RowExclusiveLock, ShareUpdateExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

ExclusiveLock

Acquired by `LOCK TABLE` table for `IN EXCLUSIVE MODE` statements.

Conflicts with RowShareLock, RowExclusiveLock, ShareUpdateExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

AccessExclusiveLock

Acquired by `ALTER TABLE`, `DROP TABLE`, `VACUUM FULL` and `LOCK TABLE` statements.

Conflicts with all modes (AccessShareLock, RowShareLock, RowExclusiveLock, ShareUpdateExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock).

Note

Only AccessExclusiveLock blocks `SELECT` (without `FOR UPDATE`) statement.

9.6.2. Row-level locks

Row-level locks are acquired when rows are being updated (or deleted or marked for update). Row-level locks don't affect data querying. They block writers to *the same row* only.

PostgreSQL doesn't remember any information about modified rows in memory and so has no limit to the number of rows locked at one time. However, locking a row may cause a disk write; thus, for example, `SELECT FOR UPDATE` will modify selected rows to mark them and so will result in disk writes.

In addition to table and row locks, short-term share/exclusive locks are used to control read/write access to table pages in the shared buffer pool. These locks are released immediately after a tuple is fetched or updated. Application writers normally need not be concerned with page-level locks, but we mention them for completeness.

9.7. Locking and Indexes

Though PostgreSQL provides nonblocking read/write access to table data, nonblocking read/write access is not currently offered for every index access method implemented in PostgreSQL.

The various index types are handled as follows:

GiST and R-Tree indexes

Share/exclusive index-level locks are used for read/write access. Locks are released after statement is done.

Hash indexes

Share/exclusive page-level locks are used for read/write access. Locks are released after page is processed.

Page-level locks provide better concurrency than index-level ones but are subject to deadlocks.

B-tree indexes

Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index tuple is fetched/inserted.

B-tree indexes provide the highest concurrency without deadlock conditions.

In short, B-tree indexes are the recommended index type for concurrent applications.

Chapter 10. Managing a Database

Although the *site administrator* is responsible for overall management of the PostgreSQL installation, some databases within the installation may be managed by another person, designated the *database administrator*. This assignment of responsibilities occurs when a database is created. A user may be assigned explicit privileges to create databases and/or to create new users. A user assigned both privileges can perform most administrative tasks within PostgreSQL, but will not by default have the same operating system privileges as the site administrator.

The *Administrator's Guide* covers these topics in more detail.

10.1. Database Creation

Databases are created by the `CREATE DATABASE` command issued from within PostgreSQL. createdb is a shell script provided to give the same functionality from the Unix command line.

The PostgreSQL backend must be running for either method to succeed, and the user issuing the command must be the PostgreSQL *superuser* or have been assigned database creation privileges by the superuser.

To create a new database named `mydb` from the command line, type

```
% createdb mydb
```

and to do the same from within psql type

```
=> CREATE DATABASE mydb;
```

If you do not have the privileges required to create a database, you will see the following:

```
ERROR:  CREATE DATABASE: Permission denied.
```

You automatically become the database administrator of the database you just created. Database names must have an alphabetic first character and are limited to 31 characters in length. PostgreSQL allows you to create any number of databases at a given site.

The *Administrator's Guide* discusses database creation in more detail, including advanced options of the `CREATE DATABASE` command.

10.2. Accessing a Database

Once you have constructed a database, you can access it by:

- Running the PostgreSQL interactive terminal program, called psql, which allows you to interactively enter, edit, and execute SQL commands.
- Using an existing graphical frontend tool like PgAccess or ApplixWare (via ODBC) to create and manipulate a database. These possibilities are not covered in this tutorial.
- Writing a custom application, using one of the several available language bindings. These possibilities are discussed further in *The PostgreSQL Programmer's Guide*.

You probably want to start up psql, to try out the examples in this manual. It can be activated for the mydb database by typing the command:

```
% psql mydb
```

You will be greeted with the following message:

```
Welcome to psql, 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

mydb=>
```

This prompt indicates that `psql` is listening to you and that you can type SQL queries into a work space maintained by the terminal monitor. The psql program itself responds to special commands that begin with the backslash character, `\`. For example, you can get help on the syntax of various PostgreSQL SQL commands by typing:

```
mydb=> \h
```

Once you have finished entering your queries into the work space, you can pass the contents of the work space to the PostgreSQL server by typing:

```
mydb=> \g
```

This tells the server to process the query. If you terminate your query with a semicolon, the `\g` is not necessary. psql will automatically process semicolon terminated queries. To read queries from a file, say `myFile`, instead of entering them interactively, type:

```
mydb=> \i myFile
```

To get out of psql and return to Unix, type

```
mydb=> \q
```

and psql will quit and return you to your command shell. (For more escape codes, type `\?` at the `psql` prompt.) White space (i.e., spaces, tabs and newlines) may be used freely in SQL queries. Single-line comments are denoted by `--`. Everything after the dashes up to the end of the line is ignored. Multiple-line comments, and comments within a line, are denoted by `/* ... */`.

10.3. Destroying a Database

If you are the owner of the database mydb, you can destroy it using the SQL command

```
=> DROP DATABASE mydb;
```

or the Unix shell script

```
% dropdb mydb
```

This action physically removes all of the Unix files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

Chapter 11. Performance Tips

Query performance can be affected by many things. Some of these can be manipulated by the user, while others are fundamental to the underlying design of the system. This chapter provides some hints about understanding and tuning PostgreSQL performance.

11.1. Using `EXPLAIN`

PostgreSQL devises a *query plan* for each query it is given. Choosing the right plan to match the query structure and the properties of the data is absolutely critical for good performance. You can use the `EXPLAIN` command to see what query plan the system creates for any query. Plan-reading is an art that deserves an extensive tutorial, which this is not; but here is some basic information.

The numbers that are currently quoted by `EXPLAIN` are:

- Estimated start-up cost (time expended before output scan can start, e.g., time to do the sorting in a SORT node).
- Estimated total cost (if all tuples are retrieved, which they may not be --- a query with a LIMIT will stop short of paying the total cost, for example).
- Estimated number of rows output by this plan node (again, without regard for any LIMIT).
- Estimated average width (in bytes) of rows output by this plan node.

The costs are measured in units of disk page fetches. (CPU effort estimates are converted into disk-page units using some fairly arbitrary fudge-factors. If you want to experiment with these factors, see the list of run-time configuration parameters in the *Administrator's Guide*.)

It's important to note that the cost of an upper-level node includes the cost of all its child nodes. It's also important to realize that the cost only reflects things that the planner/optimizer cares about. In particular, the cost does not consider the time spent transmitting result tuples to the frontend --- which could be a pretty dominant factor in the true elapsed time, but the planner ignores it because it cannot change it by altering the plan. (Every correct plan will output the same tuple set, we trust.)

Rows output is a little tricky because it is *not* the number of rows processed/scanned by the query --- it is usually less, reflecting the estimated selectivity of any WHERE-clause constraints that are being applied at this node. Ideally the top-level rows estimate will approximate the number of rows actually returned, updated, or deleted by the query.

Here are some examples (using the regress test database after a vacuum analyze, and 7.2 development sources):

```
regression=# EXPLAIN SELECT * FROM tenk1;
NOTICE:  QUERY PLAN:

Seq Scan on tenk1  (cost=0.00..333.00 rows=10000 width=148)
```

This is about as straightforward as it gets. If you do

```
SELECT * FROM pg_class WHERE relname = 'tenk1';
```

you will find out that tenk1 has 233 disk pages and 10000 tuples. So the cost is estimated at 233 page reads, defined as 1.0 apiece, plus 10000 * cpu_tuple_cost which is currently 0.01 (try show cpu_tuple_cost).

Now let's modify the query to add a qualification clause:

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;
NOTICE:  QUERY PLAN:

Seq Scan on tenk1  (cost=0.00..358.00 rows=1007 width=148)
```

The estimate of output rows has gone down because of the WHERE clause. However, the scan will still have to visit all 10000 rows, so the cost hasn't decreased; in fact it has gone up a bit to reflect the extra CPU time spent checking the WHERE condition.

The actual number of rows this query would select is 1000, but the estimate is only approximate. If you try to duplicate this experiment, you will probably get a slightly different estimate; moreover, it will change after each ANALYZE command, because the statistics produced by ANALYZE are taken from a randomized sample of the table.

Modify the query to restrict the qualification even more:

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50;
NOTICE:  QUERY PLAN:

Index Scan using tenk1_unique1 on tenk1  (cost=0.00..181.09 rows=49
 width=148)
```

and you will see that if we make the WHERE condition selective enough, the planner will eventually decide that an index scan is cheaper than a sequential scan. This plan will only have to visit 50 tuples because of the index, so it wins despite the fact that each individual fetch is more expensive than reading a whole disk page sequentially.

Add another condition to the qualification:

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50 AND
regression-# stringu1 = 'xxx';
NOTICE:  QUERY PLAN:

Index Scan using tenk1_unique1 on tenk1  (cost=0.00..181.22 rows=1
 width=148)
```

The added clause stringu1 = 'xxx' reduces the output-rows estimate, but not the cost because we still have to visit the same set of tuples.

Let's try joining two tables, using the fields we have been discussing:

```
regression=# EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1
 < 50
regression-# AND t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Nested Loop  (cost=0.00..330.41 rows=49 width=296)
  ->  Index Scan using tenk1_unique1 on tenk1 t1
```

```
               (cost=0.00..181.09 rows=49 width=148)
  ->  Index Scan using tenk2_unique2 on tenk2 t2
               (cost=0.00..3.01 rows=1 width=148)
```

In this nested-loop join, the outer scan is the same index scan we had in the example before last, and so its cost and row count are the same because we are applying the `unique1 < 50` WHERE clause at that node. The `t1.unique2 = t2.unique2` clause is not relevant yet, so it doesn't affect row count of the outer scan. For the inner scan, the unique2 value of the current outer-scan tuple is plugged into the inner index scan to produce an index qualification like `t2.unique2 = constant`. So we get the same inner-scan plan and costs that we'd get from, say, `explain select * from tenk2 where unique2 = 42`. The costs of the loop node are then set on the basis of the cost of the outer scan, plus one repetition of the inner scan for each outer tuple (49 * 3.01, here), plus a little CPU time for join processing.

In this example the loop's output row count is the same as the product of the two scans' row counts, but that's not true in general, because in general you can have WHERE clauses that mention both relations and so can only be applied at the join point, not to either input scan. For example, if we added `WHERE ... AND t1.hundred < t2.hundred`, that would decrease the output row count of the join node, but not change either input scan.

One way to look at variant plans is to force the planner to disregard whatever strategy it thought was the winner, using the enable/disable flags for each plan type. (This is a crude tool, but useful. See also Section 11.3, “Controlling the Planner with Explicit JOINs”.)

```
regression=# set enable_nestloop = off;
SET VARIABLE
regression=# EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1
 < 50
regression-# AND t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Hash Join  (cost=181.22..564.83 rows=49 width=296)
  ->  Seq Scan on tenk2 t2
               (cost=0.00..333.00 rows=10000 width=148)
  ->  Hash  (cost=181.09..181.09 rows=49 width=148)
        ->  Index Scan using tenk1_unique1 on tenk1 t1
               (cost=0.00..181.09 rows=49 width=148)
```

This plan proposes to extract the 50 interesting rows of `tenk1` using ye same olde index scan, stash them into an in-memory hash table, and then do a sequential scan of `tenk2`, probing into the hash table for possible matches of `t1.unique2 = t2.unique2` at each `tenk2` tuple. The cost to read `tenk1` and set up the hash table is entirely start-up cost for the hash join, since we won't get any tuples out until we can start reading `tenk2`. The total time estimate for the join also includes a hefty charge for CPU time to probe the hash table 10000 times. Note, however, that we are NOT charging 10000 times 181.09; the hash table setup is only done once in this plan type.

It is possible to check on the accuracy of the planner's estimated costs by using EXPLAIN ANALYZE. This command actually executes the query, and then displays the true runtime accumulated within each plan node along with the same estimated costs that a plain EXPLAIN shows. For example, we might get a result like this:

```
regression=# EXPLAIN ANALYZE
regression-# SELECT * FROM tenk1 t1, tenk2 t2
regression-# WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;
```

```
NOTICE:  QUERY PLAN:

Nested Loop  (cost=0.00..330.41 rows=49 width=296) (actual
 time=1.31..28.90 rows=50 loops=1)
  ->  Index Scan using tenk1_unique1 on tenk1 t1
                (cost=0.00..181.09 rows=49 width=148) (actual
 time=0.69..8.84 rows=50 loops=1)
  ->  Index Scan using tenk2_unique2 on tenk2 t2
                (cost=0.00..3.01 rows=1 width=148) (actual
 time=0.28..0.31 rows=1 loops=50)
Total runtime: 30.67 msec
```

Note that the “actual time” values are in milliseconds of real time, whereas the “cost” estimates are expressed in arbitrary units of disk fetches; so they are unlikely to match up. The thing to pay attention to is the ratios.

In some query plans, it is possible for a subplan node to be executed more than once. For example, the inner index scan is executed once per outer tuple in the above nested-loop plan. In such cases, the “loops” value reports the total number of executions of the node, and the actual time and rows values shown are averages per-execution. This is done to make the numbers comparable with the way that the cost estimates are shown. Multiply by the “loops” value to get the total time actually spent in the node.

The “total runtime” shown by EXPLAIN ANALYZE includes executor startup and shutdown time, as well as time spent processing the result tuples. It does not include parsing, rewriting, or planning time. For a SELECT query, the total runtime will normally be just a little larger than the total time reported for the top-level plan node. For INSERT, UPDATE, and DELETE queries, the total runtime may be considerably larger, because it includes the time spent processing the output tuples. In these queries, the time for the top plan node essentially is the time spent computing the new tuples and/or locating the old ones, but it doesn't include the time spent making the changes.

It is worth noting that EXPLAIN results should not be extrapolated to situations other than the one you are actually testing; for example, results on a toy-sized table can't be assumed to apply to large tables. The planner's cost estimates are not linear and so it may well choose a different plan for a larger or smaller table. An extreme example is that on a table that only occupies one disk page, you'll nearly always get a sequential scan plan whether indexes are available or not. The planner realizes that it's going to take one disk page read to process the table in any case, so there's no value in expending additional page reads to look at an index.

11.2. Statistics used by the Planner

As we saw in the previous section, the query planner needs to estimate the number of rows retrieved by a query in order to make good choices of query plans. This section provides a quick look at the statistics that the system uses for these estimates.

One component of the statistics is the total number of entries in each table and index, as well as the number of disk blocks occupied by each table and index. This information is kept in `pg_class`'s `reltuples` and `relpages` columns. We can look at it with queries similar to this one:

```
regression=# select relname, relkind, reltuples, relpages from
 pg_class
regression-# 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 rows)
```

Here we can see that `tenk1` contains 10000 rows, as do its indexes, but the indexes are (unsurprisingly) much smaller than the table.

For efficiency reasons, `reltuples` and `relpages` are not updated on-the-fly, and so they usually contain only approximate values (which is good enough for the planner's purposes). They are initialized with dummy values (presently 1000 and 10 respectively) when a table is created. They are updated by certain commands, presently `VACUUM`, `ANALYZE`, and `CREATE INDEX`. A stand-alone `ANALYZE`, that is one not part of `VACUUM`, generates an approximate `reltuples` value since it does not read every row of the table.

Most queries retrieve only a fraction of the rows in a table, due to having WHERE clauses that restrict the rows to be examined. The planner thus needs to make an estimate of the *selectivity* of WHERE clauses, that is, the fraction of rows that match each clause of the WHERE condition. The information used for this task is stored in the `pg_statistic` system catalog. Entries in `pg_statistic` are updated by `ANALYZE` and `VACUUM ANALYZE` commands, and are always approximate even when freshly updated.

Rather than look at `pg_statistic` directly, it's better to look at its view `pg_stats` when examining the statistics manually. `pg_stats` is designed to be more easily readable. Furthermore, `pg_stats` is readable by all, whereas `pg_statistic` is only readable by the superuser. (This prevents unprivileged users from learning something about the contents of other people's tables from the statistics. The `pg_stats` view is restricted to show only rows about tables that the current user can read.) For example, we might do:

```
regression=# 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","I-
 880                     Ramp","Sp Railroad
 ","I- 580                        ","I- 680
 Ramp","I- 80                     Ramp","14th
     St  ","5th                       St  ","Mission
         Blvd","I- 880                        "}
 thepath |         20 | {"[(-122.089,37.71),(-122.0886,37.711)]"}
(2 rows)
regression=#
```

As of PostgreSQL 7.2 the following columns exist in `pg_stats`:

Table 11.1. `pg_stats` Columns

Name	Type	Description
`tablename`	`name`	Name of table containing column
`attname`	`name`	Column described by this row

Name	Type	Description
null_frac	real	Fraction of column's entries that are NULL
avg_width	integer	Average width in bytes of column's entries
n_distinct	real	If greater than zero, the estimated number of distinct values in the column. If less than zero, the negative of the number of distinct values divided by the number of rows. (The negated form is used when ANALYZE believes that the number of distinct values is likely to increase as the table grows; the positive form is used when the column seems to have a fixed number of possible values.) For example, -1 indicates a unique column in which the number of distinct values is the same as the number of rows.
most_common_vals	text[]	A list of the most common values in the column. (Omitted if no values seem to be more common than any others.)
most_common_freqs	real[]	A list of the frequencies of the most common values, ie, number of occurrences of each divided by total number of rows.
histogram_bounds	text[]	A list of values that divide the column's values into groups of approximately equal population. The most_common_vals, if present, are omitted from the histogram calculation. (Omitted if column data type does not have a < operator, or if the most_common_vals list accounts for the entire population.)
correlation	real	Statistical correlation between physical row ordering and logical ordering of the column values. This ranges from -1 to +1. When the value is near -1 or +1, an index scan on the column will be estimated to be cheaper than when it is near zero, due to reduction of random access to the disk. (Omitted if column data type does not have a < operator.)

The maximum number of entries in the `most_common_vals` and `histogram_bounds` arrays can be set on a column-by-column basis using the `ALTER TABLE SET STATISTICS` command. The default limit is presently 10 entries. Raising the limit may allow more accurate planner estimates to be made, particularly for columns with irregular data distributions, at the price of consuming more space in `pg_statistic` and slightly more time to compute the estimates. Conversely, a lower limit may be appropriate for columns with simple data distributions.

11.3. Controlling the Planner with Explicit JOINs

Beginning with PostgreSQL 7.1 it is possible to control the query planner to some extent by using explicit JOIN syntax. To see why this matters, we first need some background.

In a simple join query, such as

```
SELECT * FROM a,b,c WHERE a.id = b.id AND b.ref = c.id;
```

the planner is free to join the given tables in any order. For example, it could generate a query plan that joins A to B, using the WHERE clause a.id = b.id, and then joins C to this joined table, using the other WHERE clause. Or it could join B to C and then join A to that result. Or it could join A to C and then join them with B --- but that would be inefficient, since the full Cartesian product of A and C would have to be formed, there being no applicable WHERE clause to allow optimization of the join. (All joins in the PostgreSQL executor happen between two input tables, so it's necessary to build up the result in one or another of these fashions.) The important point is that these different join possibilities give semantically equivalent results but may have hugely different execution costs. Therefore, the planner will explore all of them to try to find the most efficient query plan.

When a query only involves two or three tables, there aren't many join orders to worry about. But the number of possible join orders grows exponentially as the number of tables expands. Beyond ten or so input tables it's no longer practical to do an exhaustive search of all the possibilities, and even for six or seven tables planning may take an annoyingly long time. When there are too many input tables, the PostgreSQL planner will switch from exhaustive search to a *genetic* probabilistic search through a limited number of possibilities. (The switch-over threshold is set by the `GEQO_THRESHOLD` run-time parameter described in the *Administrator's Guide*.) The genetic search takes less time, but it won't necessarily find the best possible plan.

When the query involves outer joins, the planner has much less freedom than it does for plain (inner) joins. For example, consider

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id =
 b.id);
```

Although this query's restrictions are superficially similar to the previous example, the semantics are different because a row must be emitted for each row of A that has no matching row in the join of B and C. Therefore the planner has no choice of join order here: it must join B to C and then join A to that result. Accordingly, this query takes less time to plan than the previous query.

In PostgreSQL 7.1, the planner treats all explicit JOIN syntaxes as constraining the join order, even though it is not logically necessary to make such a constraint for inner joins. Therefore, although all of these queries give the same result:

```
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);
```

the second and third take less time to plan than the first. This effect is not worth worrying about for only three tables, but it can be a lifesaver with many tables.

You do not need to constrain the join order completely in order to cut search time, because it's OK to use JOIN operators in a plain FROM list. For example,

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

forces the planner to join A to B before joining them to other tables, but doesn't constrain its choices otherwise. In this example, the number of possible join orders is reduced by a factor of 5.

If you have a mix of outer and inner joins in a complex query, you might not want to constrain the planner's search for a good ordering of inner joins inside an outer join. You can't do that directly in the JOIN syntax, but you can get around the syntactic limitation by using subselects. For example,

```
SELECT * FROM d LEFT JOIN
        (SELECT * FROM a, b, c WHERE ...) AS ss
        ON (...);
```

Here, joining D must be the last step in the query plan, but the planner is free to consider various join orders for A,B,C.

Constraining the planner's search in this way is a useful technique both for reducing planning time and for directing the planner to a good query plan. If the planner chooses a bad join order by default, you can force it to choose a better order via JOIN syntax --- assuming that you know of a better order, that is. Experimentation is recommended.

11.4. Populating a Database

One may need to do a large number of table insertions when first populating a database. Here are some tips and techniques for making that as efficient as possible.

11.4.1. Disable Autocommit

Turn off autocommit and just do one commit at the end. (In plain SQL, this means issuing `BEGIN` at the start and `COMMIT` at the end. Some client libraries may do this behind your back, in which case you need to make sure the library does it when you want it done.) If you allow each insertion to be committed separately, PostgreSQL is doing a lot of work for each record added.

11.4.2. Use COPY FROM

Use `COPY FROM STDIN` to load all the records in one command, instead of using a series of `INSERT` commands. This reduces parsing, planning, etc. overhead a great deal. If you do this then it is not necessary to turn off autocommit, since it is only one command anyway.

11.4.3. Remove Indexes

If you are loading a freshly created table, the fastest way is to create the table, bulk-load with `COPY`, then create any indexes needed for the table. Creating an index on pre-existing data is quicker than updating it incrementally as each record is loaded.

If you are augmenting an existing table, you can `DROP INDEX`, load the table, then recreate the index. Of course, the database performance for other users may be adversely affected during the time that the index is missing. One should also think twice before dropping unique indexes, since the error checking afforded by the unique constraint will be lost while the index is missing.

11.4.4. ANALYZE Afterwards

It's a good idea to run `ANALYZE` or `VACUUM ANALYZE` anytime you've added or updated a lot of data, including just after initially populating a table. This ensures that the planner has up-to-date statistics about the table. With no statistics or obsolete statistics, the planner may make poor choices of query plans, leading to bad performance on queries that use your table.

Appendix A. Date/Time Support

PostgreSQL uses an internal heuristic parser for all date/time support. Dates and times are input as strings, and are broken up into distinct fields with a preliminary determination of what kind of information may be in the field. Each field is interpreted and either assigned a numeric value, ignored, or rejected. The parser contains internal lookup tables for all textual fields, including months, days of the week, and time zones.

This appendix includes information on the content of these lookup tables and describes the steps used by the parser to decode dates and times.

A.1. Date/Time Keywords

Table A.1. Month Abbreviations

Month	Abbreviations
April	Apr
August	Aug
December	Dec
February	Feb
January	Jan
July	Jul
June	Jun
March	Mar
November	Nov
October	Oct
September	Sep, Sept

Note

The month `May` has no explicit abbreviation, for obvious reasons.

Table A.2. Day of the Week Abbreviations

Day	Abbreviation
Sunday	Sun
Monday	Mon
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri
Saturday	Sat

Table A.3. PostgreSQL Field Modifiers

Identifier	Description
ABSTIME	Keyword ignored

Identifier	Description
AM	Time is before 12:00
AT	Keyword ignored
JULIAN, JD, J	Next field is Julian Day
ON	Keyword ignored
PM	Time is on or after after 12:00
T	Next field is time

The keyword `ABSTIME` is ignored for historical reasons; in very old releases of PostgreSQL invalid `ABSTIME` fields were emitted as "Invalid Abstime". This is no longer the case however and this keyword will likely be dropped in a future release.

A.2. Time Zones

PostgreSQL contains internal tabular information for time zone decoding, since there is no *nix standard system interface to provide access to general, cross-timezone information. The underlying OS *is* used to provide time zone information for *output*, however.

The following table of time zones recognized by PostgreSQL is organized by time zone offset from UTC, rather than alphabetically; this is intended to facilitate matching local usage with recognized abbreviations for cases where these might differ.

Table A.4. PostgreSQL Recognized Time Zones

Time Zone	Offset from UTC	Description
NZDT	+13:00	New Zealand Daylight 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 Savings Time
SADT	+10:30	South Australian Daylight Time
AEST	+10:00	Australia Eastern Standard Time
EAST	+10:00	East Australian Standard Time
GST	+10:00	Guam Standard Time, USSR 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

Time Zone	Offset from UTC	Description
JST	+09:00	Japan Standard Time,USSR Zone 8
KST	+09:00	Korea Standard Time
MHT	+09:00	Kwajalein Time
WDT	+09:00	West Australian Daylight Time
MT	+08:30	Moluccas Time
AWST	+08:00	Australia Western Standard Time
CCT	+08:00	China Coastal Time
WADT	+08:00	West Australian Daylight 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
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	Afganistan Time
EAST	+04:00	Antananarivo Savings Time
MUT	+04:00	Mauritius Island Time
RET	+04:00	Reunion Island Time
SCT	+04:00	Mahe Island Time
IT	+03:30	Iran Time
EAT	+03:00	Antananarivo, Comoro Time
BT	+03:00	Baghdad Time
EETDST	+03:00	Eastern Europe Daylight Savings Time
HMT	+03:00	Hellas Mediterranean Time (?)
BDST	+02:00	British Double Standard Time
CEST	+02:00	Central European Savings Time
CETDST	+02:00	Central European Daylight Savings Time
EET	+02:00	Eastern Europe, USSR Zone 1
FWT	+02:00	French Winter Time
IST	+02:00	Israel Standard Time
MEST	+02:00	Middle Europe Summer Time
METDST	+02:00	Middle Europe Daylight Time

Time Zone	Offset from UTC	Description
SST	+02:00	Swedish Summer Time
BST	+01:00	British Summer Time
CET	+01:00	Central European Time
DNT	+01:00	*Dansk Normal Tid*
FST	+01:00	French Summer Time
MET	+01:00	Middle Europe Time
MEWT	+01:00	Middle Europe Winter Time
MEZ	+01:00	Middle Europe Zone
NOR	+01:00	Norway Standard Time
SET	+01:00	Seychelles Time
SWT	+01:00	Swedish Winter Time
WETDST	+01:00	Western Europe Daylight Savings Time
GMT	+00:00	Greenwich Mean Time
UT	+00:00	Universal Time
UTC	+00:00	Universal Time, Coordinated
Z	+00:00	Same as UTC
ZULU	+00:00	Same as UTC
WET	+00:00	Western Europe
WAT	-01:00	West Africa Time
NDT	-02:30	Newfoundland Daylight Time
ADT	-03:00	Atlantic Daylight Time
AWT	-03:00	(unknown)
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
ACT	-05:00	Atlantic/Porto Acre Standard Time
EDT	-04:00	Eastern Daylight Time
CDT	-05:00	Central Daylight Time
EST	-05:00	Eastern Standard Time
CST	-06:00	Central Standard Time
MDT	-06:00	Mountain Daylight Time
MST	-07:00	Mountain Standard Time
PDT	-07:00	Pacific Daylight Time
AKDT	-08:00	Alaska Daylight Time
PST	-08:00	Pacific Standard Time

Time Zone	Offset from UTC	Description
YDT	-08:00	Yukon Daylight Time
AKST	-09:00	Alaska Standard Time
HDT	-09:00	Hawaii/Alaska Daylight Time
YST	-09:00	Yukon Standard 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

A.2.1. Australian Time Zones

Australian time zones and their naming variants account for fully one quarter of all time zones in the PostgreSQL time zone lookup table. There are two naming conflicts with time zones commonly used in the United States, `CST` and `EST`.

If the runtime option `AUSTRALIAN_TIMEZONES` is set then `CST`, `EST`, and `SAT` will be interpreted as Australian timezone names. Without this option, `CST` and `EST` are taken as American timezone names, while `SAT` is interpreted as a noise word indicating `Saturday`.

Table A.5. PostgreSQL Australian Time Zones

Time Zone	Offset from UTC	Description
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.2.2. Date/Time Input Interpretation

The date/time types are all decoded using a common set of routines.

Date/Time Input Interpretation

1. Break the input string into tokens and categorize each token as a string, time, time zone, or number.

    a. If the numeric token contains a colon (":"), this is a time string. Include all subsequent digits and colons.

    b. If the numeric token contains a dash ("-"), slash ("/"), or two or more dots ("."), this is a date string which may have a text month.

    c. If the token is numeric only, then it is either a single field or an ISO-8601 concatenated date (e.g. `19990113` for January 13, 1999) or time (e.g. 141516 for 14:15:16).

    d. If the token starts with a plus ("+") or minus ("-"), then it is either a time zone or a special field.

2. If the token is a text string, match up with possible strings.

    a. Do a binary-search table lookup for the token as either a special string (e.g. `today`), day (e.g. `Thursday`), month (e.g. `January`), or noise word (e.g. `at`, `on`).

    Set field values and bit mask for fields. For example, set year, month, day for `today`, and additionally hour, minute, second for `now`.

    b. If not found, do a similar binary-search table lookup to match the token with a time zone.

    c. If not found, throw an error.

3. The token is a number or number field.

    a. If there are more than 4 digits, and if no other date fields have been previously read, then interpret as a "concatenated date" (e.g. `19990118`). 8 and 6 digits are interpreted as year, month, and day, while 7 and 5 digits are interpreted as year, day of year, respectively.

    b. If the token is three digits and a year has already been decoded, then interpret as day of year.

    c. If four or six digits and a year has already been read, then interpret as a time.

    d. If four or more digits, then interpret as a year.

    e. If in European date mode, and if the day field has not yet been read, and if the value is less than or equal to 31, then interpret as a day.

    f. If the month field has not yet been read, and if the value is less than or equal to 12, then interpret as a month.

    g. If the day field has not yet been read, and if the value is less than or equal to 31, then interpret as a day.

    h. If two digits or four or more digits, then interpret as a year.

    i. Otherwise, throw an error.

4. If BC has been specified, negate the year and add one for internal storage (there is no year zero in the Gregorian calendar, so numerically `1BC` becomes year zero).

5. If BC was not specified, and if the year field was two digits in length, then adjust the year to 4 digits. If the field was less than 70, then add 2000; otherwise, add 1900.

Tip

Gregorian years 1-99AD may be entered by using 4 digits with leading zeros (e.g. 0099 is 99AD). Previous versions of PostgreSQL accepted years with three digits and with single digits, but as of version 7.0 the rules have been tightened up to reduce the possibility of ambiguity.

A.3. History of Units

Note

Contributed by José Soares (<jose@sferacarta.com>)

The Julian Day was invented by the French scholar Joseph Justus Scaliger (1540-1609) and probably takes its name from the Scaliger's father, the Italian scholar Julius Caesar Scaliger (1484-1558). Astronomers have used the Julian period to assign a unique number to every day since 1 January 4713 BC. This is the so-called Julian Day (JD). JD 0 designates the 24 hours from noon UTC on 1 January 4713 BC to noon UTC on 2 January 4713 BC.

“Julian Day” is different from “Julian Date”. The Julian calendar was introduced by Julius Caesar in 45 BC. It was in common use until the 1582, when countries started changing to the Gregorian calendar. In the Julian calendar, the tropical year is approximated as 365 1/4 days = 365.25 days. This gives an error of about 1 day in 128 years. The accumulating calendar error prompted Pope Gregory XIII to reform the calendar in accordance with instructions from the Council of Trent.

In the Gregorian calendar, the tropical year is approximated as 365 + 97 / 400 days = 365.2425 days. Thus it takes approximately 3300 years for the tropical year to shift one day with respect to the Gregorian calendar.

The approximation 365+97/400 is achieved by having 97 leap years every 400 years, using the following rules:

Every year divisible by 4 is a leap year.
However, every year divisible by 100 is not a leap year.
However, every year divisible by 400 is a leap year after all.
So, 1700, 1800, 1900, 2100, and 2200 are not leap years. But 1600, 2000, and 2400 are leap years. By contrast, in the older Julian calendar only years divisible by 4 are leap years.

The papal bull of February 1582 decreed that 10 days should be dropped from October 1582 so that 15 October should follow immediately after 4 October. This was observed in Italy, Poland, Portugal, and Spain. Other Catholic countries followed shortly after, but Protestant countries were reluctant to change, and the Greek orthodox countries didn't change until the start of this century. The reform was observed by Great Britain and Dominions (including what is now the USA) in 1752. Thus 2 Sep 1752 was followed by 14 Sep 1752. This is why Unix systems have cal produce the following:

```
% cal 9 1752
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Note

SQL92 states that “Within the definition of a ‘datetime literal’, the ‘datetime value’s are constrained by the natural rules for dates and times according to the Gregorian calendar”. Dates between 1752-09-03 and 1752-09-13, although eliminated in some countries by Papal fiat, conform to “natural rules” and are hence valid dates.

Different calendars have been developed in various parts of the world, many predating the Gregorian system. For example, the beginnings of the Chinese calendar can be traced back to the 14th century BC. Legend has it that the Emperor Huangdi invented the calendar in 2637 BC. The People's Republic of China uses the Gregorian calendar for civil purposes. Chinese calendar is used for determining festivals.

Appendix B. SQL Key Words

Table B.1, “SQL Key Words” lists all tokens that are key words in the SQL standard and in PostgreSQL 7.2.8. Background information can be found in Section 1.1.1, “Identifiers and Key Words”.

SQL distinguishes between *reserved* and *non-reserved* key words. According to the standard, reserved key words are the only real key words; they are never allowed as identifiers. Non-reserved key words only have a special meaning in particular contexts and can be used as identifiers in other contexts. Most non-reserved key words are actually the names of built-in tables and functions specified by SQL. The concept of non-reserved key words essentially only exists to declare that some predefined meaning is attached to a word in some contexts.

In the PostgreSQL parser life is a bit more complicated. There are several different classes of tokens ranging from those that can never be used as an identifier to those that have absolutely no special status in the parser as compared to an ordinary identifier. (The latter is usually the case for functions specified by SQL.) Even reserved key words are not completely reserved in PostgreSQL, but can be used as column labels (for example, `SELECT 55 AS CHECK`, even though `CHECK` is a reserved key word).

In Table B.1, “SQL Key Words” in the column for PostgreSQL we classify as “non-reserved” those key words that are explicitly known to the parser but are allowed in most or all contexts where an identifier is expected. Some key words that are otherwise non-reserved cannot be used as function or data type names and are marked accordingly. (Most of these words represent built-in functions or data types with special syntax. The function or type is still available but it cannot be redefined by the user.) Labeled “reserved” are those tokens that are only allowed as “AS” column label names (and perhaps in very few other contexts). Some reserved key words are allowable as names for functions; this is also shown in the table.

As a general rule, if you get spurious parser errors for commands that contain any of the listed key words as an identifier you should try to quote the identifier to see if the problem goes away.

It is important to understand before studying Table B.1, “SQL Key Words” that the fact that a key word is not reserved in PostgreSQL does not mean that the feature related to the word is not implemented. Conversely, the presence of a key word does not indicate the existence of a feature.

Table B.1. SQL Key Words

Key Word	PostgreSQL	SQL 99	SQL 92
ABORT	non-reserved		
ABS		non-reserved	
ABSOLUTE	non-reserved	reserved	reserved
ACCESS	non-reserved		
ACTION	non-reserved	reserved	reserved
ADA		non-reserved	non-reserved
ADD	non-reserved	reserved	reserved
ADMIN		reserved	
AFTER	non-reserved	reserved	
AGGREGATE	non-reserved	reserved	
ALIAS		reserved	
ALL	reserved	reserved	reserved
ALLOCATE		reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
ALTER	non-reserved	reserved	reserved
ANALYSE	reserved		
ANALYZE	reserved		
AND	reserved	reserved	reserved
ANY	reserved	reserved	reserved
ARE		reserved	reserved
ARRAY		reserved	
AS	reserved	reserved	reserved
ASC	reserved	reserved	reserved
ASENSITIVE		non-reserved	
ASSERTION		reserved	reserved
ASSIGNMENT		non-reserved	
ASYMMETRIC		non-reserved	
AT	non-reserved	reserved	reserved
ATOMIC		non-reserved	
AUTHORIZATION	non-reserved	reserved	reserved
AVG		non-reserved	reserved
BACKWARD	non-reserved		
BEFORE	non-reserved	reserved	
BEGIN	non-reserved	reserved	reserved
BETWEEN	reserved (can be function)	non-reserved	reserved
BINARY	reserved (can be function)	reserved	
BIT	non-reserved (cannot be function or type)	reserved	reserved
BITVAR		non-reserved	
BIT_LENGTH		non-reserved	reserved
BLOB		reserved	
BOOLEAN		reserved	
BOTH	reserved	reserved	reserved
BREADTH		reserved	
BY	non-reserved	reserved	reserved
C		non-reserved	non-reserved
CACHE	non-reserved		
CALL		reserved	
CALLED		non-reserved	
CARDINALITY		non-reserved	
CASCADE	non-reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
CASCADED		reserved	reserved
CASE	reserved	reserved	reserved
CAST	reserved	reserved	reserved
CATALOG		reserved	reserved
CATALOG_NAME		non-reserved	non-reserved
CHAIN	non-reserved	non-reserved	
CHAR	non-reserved (cannot be function or type)	reserved	reserved
CHARACTER	non-reserved (cannot be function or type)	reserved	reserved
CHARACTERISTICS	non-reserved		
CHARACTER_LENGTH		non-reserved	reserved
CHARACTER_SET_CATALOG		non-reserved	non-reserved
CHARACTER_SET_NAME		non-reserved	non-reserved
CHARACTER_SET_SCHEMA		non-reserved	non-reserved
CHAR_LENGTH		non-reserved	reserved
CHECK	reserved	reserved	reserved
CHECKED		non-reserved	
CHECKPOINT	non-reserved		
CLASS		reserved	
CLASS_ORIGIN		non-reserved	non-reserved
CLOB		reserved	
CLOSE	non-reserved	reserved	reserved
CLUSTER	non-reserved		
COALESCE	non-reserved (cannot be function or type)	non-reserved	reserved
COBOL		non-reserved	non-reserved
COLLATE	reserved	reserved	reserved
COLLATION		reserved	reserved
COLLATION_CATALOG		non-reserved	non-reserved
COLLATION_NAME		non-reserved	non-reserved
COLLATION_SCHEMA		non-reserved	non-reserved
COLUMN	reserved	reserved	reserved
COLUMN_NAME		non-reserved	non-reserved
COMMAND_FUNCTION		non-reserved	non-reserved
COMMAND_FUNCTION_CODE		non-reserved	
COMMENT	non-reserved		
COMMIT	non-reserved	reserved	reserved
COMMITTED	non-reserved	non-reserved	non-reserved

Key Word	PostgreSQL	SQL 99	SQL 92
COMPLETION		reserved	
CONDITION_NUMBER		non-reserved	non-reserved
CONNECT		reserved	reserved
CONNECTION		reserved	reserved
CONNECTION_NAME		non-reserved	non-reserved
CONSTRAINT	reserved	reserved	reserved
CONSTRAINTS	non-reserved	reserved	reserved
CONSTRAINT_CATALOG		non-reserved	non-reserved
CONSTRAINT_NAME		non-reserved	non-reserved
CONSTRAINT_SCHEMA		non-reserved	non-reserved
CONSTRUCTOR		reserved	
CONTAINS		non-reserved	
CONTINUE		reserved	reserved
CONVERT		non-reserved	reserved
COPY	non-reserved		
CORRESPONDING		reserved	reserved
COUNT		non-reserved	reserved
CREATE	non-reserved	reserved	reserved
CREATEDB	non-reserved		
CREATEUSER	non-reserved		
CROSS	reserved (can be function)	reserved	reserved
CUBE		reserved	
CURRENT		reserved	reserved
CURRENT_DATE	reserved	reserved	reserved
CURRENT_PATH		reserved	
CURRENT_ROLE		reserved	
CURRENT_TIME	reserved	reserved	reserved
CURRENT_TIMESTAMP	reserved	reserved	reserved
CURRENT_USER	reserved	reserved	reserved
CURSOR	non-reserved	reserved	reserved
CURSOR_NAME		non-reserved	non-reserved
CYCLE	non-reserved	reserved	
DATA		reserved	non-reserved
DATABASE	non-reserved		
DATE		reserved	reserved
DATETIME_INTERVAL_CODE		non-reserved	non-reserved
DATETIME_INTERVAL_PRECISION		non-reserved	non-reserved

Key Word	PostgreSQL	SQL 99	SQL 92
DAY	non-reserved	reserved	reserved
DEALLOCATE		reserved	reserved
DEC	non-reserved (cannot be function or type)	reserved	reserved
DECIMAL	non-reserved (cannot be function or type)	reserved	reserved
DECLARE	non-reserved	reserved	reserved
DEFAULT	reserved	reserved	reserved
DEFERRABLE	reserved	reserved	reserved
DEFERRED	non-reserved	reserved	reserved
DEFINED		non-reserved	
DEFINER		non-reserved	
DELETE	non-reserved	reserved	reserved
DELIMITERS	non-reserved		
DEPTH		reserved	
DEREF		reserved	
DESC	reserved	reserved	reserved
DESCRIBE		reserved	reserved
DESCRIPTOR		reserved	reserved
DESTROY		reserved	
DESTRUCTOR		reserved	
DETERMINISTIC		reserved	
DIAGNOSTICS		reserved	reserved
DICTIONARY		reserved	
DISCONNECT		reserved	reserved
DISPATCH		non-reserved	
DISTINCT	reserved	reserved	reserved
DO	reserved		
DOMAIN		reserved	reserved
DOUBLE	non-reserved	reserved	reserved
DROP	non-reserved	reserved	reserved
DYNAMIC		reserved	
DYNAMIC_FUNCTION		non-reserved	non-reserved
DYNAMIC_FUNCTION_CODE		non-reserved	
EACH	non-reserved	reserved	
ELSE	reserved	reserved	reserved
ENCODING	non-reserved		
ENCRYPTED	non-reserved		

Key Word	PostgreSQL	SQL 99	SQL 92
END	reserved	reserved	reserved
END-EXEC		reserved	reserved
EQUALS		reserved	
ESCAPE	non-reserved	reserved	reserved
EVERY		reserved	
EXCEPT	reserved	reserved	reserved
EXCEPTION		reserved	reserved
EXCLUSIVE	non-reserved		
EXEC		reserved	reserved
EXECUTE	non-reserved	reserved	reserved
EXISTING		non-reserved	
EXISTS	non-reserved (cannot be function or type)	non-reserved	reserved
EXPLAIN	non-reserved		
EXTERNAL		reserved	reserved
EXTRACT	non-reserved (cannot be function or type)	non-reserved	reserved
FALSE	reserved	reserved	reserved
FETCH	non-reserved	reserved	reserved
FINAL		non-reserved	
FIRST		reserved	reserved
FLOAT	non-reserved (cannot be function or type)	reserved	reserved
FOR	reserved	reserved	reserved
FORCE	non-reserved		
FOREIGN	reserved	reserved	reserved
FORTRAN		non-reserved	non-reserved
FORWARD	non-reserved		
FOUND		reserved	reserved
FREE		reserved	
FREEZE	reserved (can be function)		
FROM	reserved	reserved	reserved
FULL	reserved (can be function)	reserved	reserved
FUNCTION	non-reserved	reserved	
G		non-reserved	
GENERAL		reserved	
GENERATED		non-reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
GET		reserved	reserved
GLOBAL	non-reserved	reserved	reserved
GO		reserved	reserved
GOTO		reserved	reserved
GRANT	non-reserved	reserved	reserved
GRANTED		non-reserved	
GROUP	reserved	reserved	reserved
GROUPING		reserved	
HANDLER	non-reserved		
HAVING	reserved	reserved	reserved
HIERARCHY		non-reserved	
HOLD		non-reserved	
HOST		reserved	
HOUR	non-reserved	reserved	reserved
IDENTITY		reserved	reserved
IGNORE		reserved	
ILIKE	reserved (can be function)		
IMMEDIATE	non-reserved	reserved	reserved
IMPLEMENTATION		non-reserved	
IN	reserved (can be function)	reserved	reserved
INCREMENT	non-reserved		
INDEX	non-reserved		
INDICATOR		reserved	reserved
INFIX		non-reserved	
INHERITS	non-reserved		
INITIALIZE		reserved	
INITIALLY	reserved	reserved	reserved
INNER	reserved (can be function)	reserved	reserved
INOUT	non-reserved	reserved	
INPUT		reserved	reserved
INSENSITIVE	non-reserved	non-reserved	reserved
INSERT	non-reserved	reserved	reserved
INSTANCE		non-reserved	
INSTANTIABLE		non-reserved	
INSTEAD	non-reserved		
INT		reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
INTEGER		reserved	reserved
INTERSECT	reserved	reserved	reserved
INTERVAL	non-reserved (cannot be function or type)	reserved	reserved
INTO	reserved	reserved	reserved
INVOKER		non-reserved	
IS	reserved (can be function)	reserved	reserved
ISNULL	reserved (can be function)		
ISOLATION	non-reserved	reserved	reserved
ITERATE		reserved	
JOIN	reserved (can be function)	reserved	reserved
K		non-reserved	
KEY	non-reserved	reserved	reserved
KEY_MEMBER		non-reserved	
KEY_TYPE		non-reserved	
LANCOMPILER	non-reserved		
LANGUAGE	non-reserved	reserved	reserved
LARGE		reserved	
LAST		reserved	reserved
LATERAL		reserved	
LEADING	reserved	reserved	reserved
LEFT	reserved (can be function)	reserved	reserved
LENGTH		non-reserved	non-reserved
LESS		reserved	
LEVEL	non-reserved	reserved	reserved
LIKE	reserved (can be function)	reserved	reserved
LIMIT	reserved	reserved	
LISTEN	non-reserved		
LOAD	non-reserved		
LOCAL	non-reserved	reserved	reserved
LOCALTIME		reserved	
LOCALTIMESTAMP		reserved	
LOCATION	non-reserved		
LOCATOR		reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
LOCK	non-reserved		
LOWER		non-reserved	reserved
M		non-reserved	
MAP		reserved	
MATCH	non-reserved	reserved	reserved
MAX		non-reserved	reserved
MAXVALUE	non-reserved		
MESSAGE_LENGTH		non-reserved	non-reserved
MESSAGE_OCTET_LENGTH		non-reserved	non-reserved
MESSAGE_TEXT		non-reserved	non-reserved
METHOD		non-reserved	
MIN		non-reserved	reserved
MINUTE	non-reserved	reserved	reserved
MINVALUE	non-reserved		
MOD		non-reserved	
MODE	non-reserved		
MODIFIES		reserved	
MODIFY		reserved	
MODULE		reserved	reserved
MONTH	non-reserved	reserved	reserved
MORE		non-reserved	non-reserved
MOVE	non-reserved		
MUMPS		non-reserved	non-reserved
NAME		non-reserved	non-reserved
NAMES	non-reserved	reserved	reserved
NATIONAL	non-reserved	reserved	reserved
NATURAL	reserved (can be function)	reserved	reserved
NCHAR	non-reserved (cannot be function or type)	reserved	reserved
NCLOB		reserved	
NEW	reserved	reserved	
NEXT	non-reserved	reserved	reserved
NO	non-reserved	reserved	reserved
NOCREATEDB	non-reserved		
NOCREATEUSER	non-reserved		
NONE	non-reserved (cannot be function or type)	reserved	
NOT	reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
NOTHING	non-reserved		
NOTIFY	non-reserved		
NOTNULL	reserved (can be function)		
NULL	reserved	reserved	reserved
NULLABLE		non-reserved	non-reserved
NULLIF	non-reserved (cannot be function or type)	non-reserved	reserved
NUMBER		non-reserved	non-reserved
NUMERIC	non-reserved (cannot be function or type)	reserved	reserved
OBJECT		reserved	
OCTET_LENGTH		non-reserved	reserved
OF	non-reserved	reserved	reserved
OFF	reserved	reserved	
OFFSET	reserved		
OIDS	non-reserved		
OLD	reserved	reserved	
ON	reserved	reserved	reserved
ONLY	reserved	reserved	reserved
OPEN		reserved	reserved
OPERATION		reserved	
OPERATOR	non-reserved		
OPTION	non-reserved	reserved	reserved
OPTIONS		non-reserved	
OR	reserved	reserved	reserved
ORDER	reserved	reserved	reserved
ORDINALITY		reserved	
OUT	non-reserved	reserved	
OUTER	reserved (can be function)	reserved	reserved
OUTPUT		reserved	reserved
OVERLAPS	reserved (can be function)	non-reserved	reserved
OVERLAY		non-reserved	
OVERRIDING		non-reserved	
OWNER	non-reserved		
PAD		reserved	reserved
PARAMETER		reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
PARAMETERS		reserved	
PARAMETER_MODE		non-reserved	
PARAMETER_NAME		non-reserved	
PARAMETER_ORDINAL_POSITION		non-reserved	
PARAMETER_SPECIFIC_CATALOG		non-reserved	
PARAMETER_SPECIFIC_NAME		non-reserved	
PARAMETER_SPECIFIC_SCHEMA		non-reserved	
PARTIAL	non-reserved	reserved	reserved
PASCAL		non-reserved	non-reserved
PASSWORD	non-reserved		
PATH	non-reserved	reserved	
PENDANT	non-reserved		
PLI		non-reserved	non-reserved
POSITION	non-reserved (cannot be function or type)	non-reserved	reserved
POSTFIX		reserved	
PRECISION	non-reserved	reserved	reserved
PREFIX		reserved	
PREORDER		reserved	
PREPARE		reserved	reserved
PRESERVE		reserved	reserved
PRIMARY	reserved	reserved	reserved
PRIOR	non-reserved	reserved	reserved
PRIVILEGES	non-reserved	reserved	reserved
PROCEDURAL	non-reserved		
PROCEDURE	non-reserved	reserved	reserved
PUBLIC	reserved (can be function)	reserved	reserved
READ	non-reserved	reserved	reserved
READS		reserved	
REAL		reserved	reserved
RECURSIVE		reserved	
REF		reserved	
REFERENCES	reserved	reserved	reserved
REFERENCING		reserved	
REINDEX	non-reserved		
RELATIVE	non-reserved	reserved	reserved
RENAME	non-reserved		

Key Word	PostgreSQL	SQL 99	SQL 92
REPEATABLE		non-reserved	non-reserved
REPLACE	non-reserved		
RESET	non-reserved		
RESTRICT	non-reserved	reserved	reserved
RESULT		reserved	
RETURN		reserved	
RETURNED_LENGTH		non-reserved	non-reserved
RETURNED_OCTET_LENGTH		non-reserved	non-reserved
RETURNED_SQLSTATE		non-reserved	non-reserved
RETURNS	non-reserved	reserved	
REVOKE	non-reserved	reserved	reserved
RIGHT	reserved (can be function)	reserved	reserved
ROLE		reserved	
ROLLBACK	non-reserved	reserved	reserved
ROLLUP		reserved	
ROUTINE		reserved	
ROUTINE_CATALOG		non-reserved	
ROUTINE_NAME		non-reserved	
ROUTINE_SCHEMA		non-reserved	
ROW	non-reserved	reserved	
ROWS		reserved	reserved
ROW_COUNT		non-reserved	non-reserved
RULE	non-reserved		
SAVEPOINT		reserved	
SCALE		non-reserved	non-reserved
SCHEMA	non-reserved	reserved	reserved
SCHEMA_NAME		non-reserved	non-reserved
SCOPE		reserved	
SCROLL	non-reserved	reserved	reserved
SEARCH		reserved	
SECOND	non-reserved	reserved	reserved
SECTION		reserved	reserved
SECURITY		non-reserved	
SELECT	reserved	reserved	reserved
SELF		non-reserved	
SENSITIVE		non-reserved	
SEQUENCE	non-reserved	reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
SERIALIZABLE	non-reserved	non-reserved	non-reserved
SERVER_NAME		non-reserved	non-reserved
SESSION	non-reserved	reserved	reserved
SESSION_USER	reserved	reserved	reserved
SET	non-reserved	reserved	reserved
SETOF	non-reserved (cannot be function or type)		
SETS		reserved	
SHARE	non-reserved		
SHOW	non-reserved		
SIMILAR		non-reserved	
SIMPLE		non-reserved	
SIZE		reserved	reserved
SMALLINT		reserved	reserved
SOME	reserved	reserved	reserved
SOURCE		non-reserved	
SPACE		reserved	reserved
SPECIFIC		reserved	
SPECIFICTYPE		reserved	
SPECIFIC_NAME		non-reserved	
SQL		reserved	reserved
SQLCODE			reserved
SQLERROR			reserved
SQLEXCEPTION		reserved	
SQLSTATE		reserved	reserved
SQLWARNING		reserved	
START	non-reserved	reserved	
STATE		reserved	
STATEMENT	non-reserved	reserved	
STATIC		reserved	
STATISTICS	non-reserved		
STDIN	non-reserved		
STDOUT	non-reserved		
STRUCTURE		reserved	
STYLE		non-reserved	
SUBCLASS_ORIGIN		non-reserved	non-reserved
SUBLIST		non-reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
SUBSTRING	non-reserved (cannot be function or type)	non-reserved	reserved
SUM		non-reserved	reserved
SYMMETRIC		non-reserved	
SYSID	non-reserved		
SYSTEM		non-reserved	
SYSTEM_USER		reserved	reserved
TABLE	reserved	reserved	reserved
TABLE_NAME		non-reserved	non-reserved
TEMP	non-reserved		
TEMPLATE	non-reserved		
TEMPORARY	non-reserved	reserved	reserved
TERMINATE		reserved	
THAN		reserved	
THEN	reserved	reserved	reserved
TIME	non-reserved (cannot be function or type)	reserved	reserved
TIMESTAMP	non-reserved (cannot be function or type)	reserved	reserved
TIMEZONE_HOUR		reserved	reserved
TIMEZONE_MINUTE		reserved	reserved
TO	reserved	reserved	reserved
TOAST	non-reserved		
TRAILING	reserved	reserved	reserved
TRANSACTION	non-reserved	reserved	reserved
TRANSACTIONS_COMMITTED		non-reserved	
TRANSACTIONS_ROLLED_BACK		non-reserved	
TRANSACTION_ACTIVE		non-reserved	
TRANSFORM		non-reserved	
TRANSFORMS		non-reserved	
TRANSLATE		non-reserved	reserved
TRANSLATION		reserved	reserved
TREAT		reserved	
TRIGGER	non-reserved	reserved	
TRIGGER_CATALOG		non-reserved	
TRIGGER_NAME		non-reserved	
TRIGGER_SCHEMA		non-reserved	
TRIM	non-reserved (cannot be function or type)	non-reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
TRUE	reserved	reserved	reserved
TRUNCATE	non-reserved		
TRUSTED	non-reserved		
TYPE	non-reserved	non-reserved	non-reserved
UNCOMMITTED		non-reserved	non-reserved
UNDER		reserved	
UNENCRYPTED	non-reserved		
UNION	reserved	reserved	reserved
UNIQUE	reserved	reserved	reserved
UNKNOWN	non-reserved	reserved	reserved
UNLISTEN	non-reserved		
UNNAMED		non-reserved	non-reserved
UNNEST		reserved	
UNTIL	non-reserved		
UPDATE	non-reserved	reserved	reserved
UPPER		non-reserved	reserved
USAGE		reserved	reserved
USER	reserved	reserved	reserved
USER_DEFINED_TYPE_CATALOG		non-reserved	
USER_DEFINED_TYPE_NAME		non-reserved	
USER_DEFINED_TYPE_SCHEMA		non-reserved	
USING	reserved	reserved	reserved
VACUUM	non-reserved		
VALID	non-reserved		
VALUE		reserved	reserved
VALUES	non-reserved	reserved	reserved
VARCHAR	non-reserved (cannot be function or type)	reserved	reserved
VARIABLE		reserved	
VARYING	non-reserved	reserved	reserved
VERBOSE	reserved (can be function)		
VERSION	non-reserved		
VIEW	non-reserved	reserved	reserved
WHEN	reserved	reserved	reserved
WHENEVER		reserved	reserved
WHERE	reserved	reserved	reserved
WITH	non-reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
WITHOUT	non-reserved	reserved	
WORK	non-reserved	reserved	reserved
WRITE		reserved	reserved
YEAR	non-reserved	reserved	reserved
ZONE	non-reserved	reserved	reserved

Bibliography

Selected references and readings for SQL and PostgreSQL.

Some white papers and technical reports from the original POSTGRES development team are available at the University of California, Berkeley, Computer Science Department web site[1]

SQL Reference Books

Reference texts for SQL features.

[bowman93] *The Practical SQL Handbook*. Bowman et al, 1996. Using Structured Query Language. Third Edition. Judith Bowman, Sandra Emerson, and Marcy Darnovsky. 0-201-44787-8. 1996. Addison-Wesley. Copyright © 1996 Addison-Wesley Longman, Inc..

[date97] *A Guide to the SQL Standard*. Date and Darwen, 1997. A user's guide to the standard database language SQL. Fourth Edition. C. J. Date and Hugh Darwen. 0-201-96426-0. 1997. Addison-Wesley. Copyright © 1997 Addison-Wesley Longman, Inc..

[date94] *An Introduction to Database Systems*. Date, 1994. Sixth Edition. C. J. Date. Volume 1. 1994. Addison-Wesley. Copyright © 1994 Addison-Wesley Longman, Inc..

[elma99] *Fundamentals of Database Systems*. 3rd Edition. Ramez Elmasri and Shamkant Navathe. 0-805-31755-4. August 1999. Addison-Wesley.

[melt93] *Understanding the New SQL*. Melton and Simon, 1993. A complete guide. Jim Melton and Alan R. Simon. 1-55860-245-3. 1993. Morgan Kaufmann. Copyright © 1993 Morgan Kaufmann Publishers, Inc..

[ull88] *Principles of Database and Knowledge*. Base Systems. Ullman, 1988. Jeffrey D. Ullman. Volume 1. Computer Science Press. 1988.

PostgreSQL-Specific Documentation

This section is for related documentation.

[sim98] *Enhancement of the ANSI SQL Implementation of PostgreSQL*. Simkovics, 1998. Stefan Simkovics. November 29, 1998. Department of Information Systems, Vienna University of Technology. Vienna, Austria.

[yu95] *The Postgres95. User Manual*. Yu and Chen, 1995. A. Yu and J. Chen. . Sept. 5, 1995. University of California. Berkeley, California.

[fong] *The design and implementation of the POSTGRES query optimizer*[2]. Zelaine Fong. University of California, Berkeley, Computer Science Department.

Proceedings and Articles

This section is for articles and newsletters.

[olson93] *Partial indexing in POSTGRES: research project*. Olson, 1993. Nels Olson. 1993. UCB Engin T7.49.1993 O676. University of California. Berkeley, California.

[1] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/
[2] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/UCB-MS-zfong.pdf

[ong90] “A Unified Framework for Version Modeling Using Production Rules in a Database System”. Ong and Goh, 1990. L. Ong and J. Goh. *ERL Technical Memorandum M90/33*. April, 1990. University of California. Berkely, California.

[rowe87] “The POSTGRES data model[3]”. Rowe and Stonebraker, 1987. L. Rowe and M. Stonebraker. VLDB Conference. Sept. 1987. Brighton, England. .

[seshadri95] “Generalized Partial Indexes[4] ”. Seshardri, 1995. P. Seshadri and A. Swami. Eleventh International Conference on Data Engineering. 6-10 March 1995. Taipeh, Taiwan. . 1995. Cat. No.95CH35724. IEEE Computer Society Press. Los Alamitos, California. 420-7.

[ston86] “The design of POSTGRES[5]”. Stonebraker and Rowe, 1986. M. Stonebraker and L. Rowe. ACM-SIGMOD Conference on Management of Data. May 1986. Washington, DC. .

[ston87a] “The design of the POSTGRES. rules system”. Stonebraker, Hanson, Hong, 1987. M. Stonebraker, E. Hanson, and C. H. Hong. IEEE Conference on Data Engineering. Feb. 1987. Los Angeles, California. .

[ston87b] “The design of the POSTGRES storage system[6]”. Stonebraker, 1987. M. Stonebraker. VLDB Conference. Sept. 1987. Brighton, England. .

[ston89] “A commentary on the POSTGRES rules system[7]”. Stonebraker et al, 1989. M. Stonebraker, M. Hearst, and S. Potamianos. *SIGMOD Record 18(3)*. Sept. 1989.

[ston89b] “The case for partial indexes[8]”. Stonebraker, M, 1989b. M. Stonebraker. *SIGMOD Record 18(4)*. 4-11. Dec. 1989.

[ston90a] “The implementation of POSTGRES[9]”. Stonebraker, Rowe, Hirohama, 1990. M. Stonebraker, L. A. Rowe, and M. Hirohama. *Transactions on Knowledge and Data Engineering 2(1)*. IEEE. March 1990.

[ston90b] “On Rules, Procedures, Caching and Views in Database Systems[10]”. Stonebraker et al, ACM, 1990. M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. ACM-SIGMOD Conference on Management of Data. June 1990. .

[3] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M87-13.pdf
[4] http://simon.cs.cornell.edu/home/praveen/papers/partindex.de95.ps.Z
[5] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M85-95.pdf
[6] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M87-06.pdf
[7] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-82.pdf
[8] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-17.pdf
[9] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M90-34.pdf
[10] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M90-36.pdf

PostgreSQL 7.2.8
Administrator's Guide

The PostgreSQL Global Development Group

PostgreSQL 7.2.8 Administrator's Guide

The PostgreSQL Global Development Group
Copyright © 1996-2001 The PostgreSQL Global Development Group

Legal Notice

PostgreSQL is Copyright © 1996-2001 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.

Table of Contents

1. Installation Instructions 1
    1.1. Short Version 1
    1.2. Requirements 1
    1.3. Getting The Source 2
    1.4. If You Are Upgrading 2
    1.5. Installation Procedure 3
    1.6. Post-Installation Setup 10
        1.6.1. Shared Libraries 10
        1.6.2. Environment Variables 10
    1.7. Supported Platforms 11
2. Installation on Windows 15
3. Server Runtime Environment 16
    3.1. The PostgreSQL user account 16
    3.2. Creating a database cluster 16
    3.3. Starting the database server 17
        3.3.1. Server Start-up Failures 18
        3.3.2. Client Connection Problems 19
    3.4. Run-time configuration 20
        3.4.1. Planner and Optimizer Tuning 21
        3.4.2. Logging and Debugging 23
        3.4.3. General operation 24
        3.4.4. WAL 29
        3.4.5. Short options 29
    3.5. Managing Kernel Resources 30
        3.5.1. Shared Memory and Semaphores 30
        3.5.2. Resource Limits 34
    3.6. Shutting down the server 35
    3.7. Secure TCP/IP Connections with SSL 36
    3.8. Secure TCP/IP Connections with SSH tunnels 37
4. Client Authentication 38
    4.1. The `pg_hba.conf` file 38
    4.2. Authentication methods 42
        4.2.1. Trust authentication 42
        4.2.2. Password authentication 43
        4.2.3. Kerberos authentication 43
        4.2.4. Ident-based authentication 44
    4.3. Authentication problems 45
5. Localization 47
    5.1. Locale Support 47
        5.1.1. Overview 47
        5.1.2. Benefits 48
        5.1.3. Problems 48
    5.2. Multibyte Support 49
        5.2.1. Enabling Multibyte Support 49
        5.2.2. Setting the Encoding 51
        5.2.3. Automatic encoding translation between server and client 51
        5.2.4. About Unicode 53
        5.2.5. What happens if the translation is not possible? 53
        5.2.6. References 53
        5.2.7. History 54
        5.2.8. WIN1250 on Windows/ODBC 55

5.3. Single-byte character set recoding 56
6. Managing Databases 57
6.1. Creating a Database 57
6.1.1. Template Databases 57
6.1.2. Alternative Locations 58
6.2. Destroying a Database 59
7. Database Users and Permissions 61
7.1. Database Users 61
7.1.1. User attributes 61
7.2. Groups 62
7.3. Privileges 62
7.4. Functions and Triggers 62
8. Routine Database Maintenance Tasks 64
8.1. General Discussion 64
8.2. Routine Vacuuming 64
8.2.1. Recovering disk space 64
8.2.2. Updating planner statistics 65
8.2.3. Preventing transaction ID wraparound failures 66
8.3. Log File Maintenance 67
9. Backup and Restore 69
9.1. SQL Dump 69
9.1.1. Restoring the dump 69
9.1.2. Using `pg_dumpall` 70
9.1.3. Large Databases 70
9.1.4. Caveats 71
9.2. File system level backup 71
9.3. Migration between releases 72
10. Monitoring Database Activity 73
10.1. Standard Unix Tools 73
10.2. Statistics Collector 74
10.2.1. Statistics Collection Configuration 74
10.2.2. Viewing Collected Statistics 74
11. Write-Ahead Logging (WAL) 78
11.1. General Description 78
11.1.1. Immediate Benefits of WAL 78
11.1.2. Future Benefits 78
11.2. Implementation 79
11.2.1. Database Recovery with WAL 79
11.3. WAL Configuration 80
12. Disk Storage 82
13. Database Failures 83
13.1. Disk Filled 83
13.2. Disk Failed 83
14. Regression Tests 84
14.1. Introduction 84
14.2. Running the Tests 84
14.3. Test Evaluation 85
14.3.1. Error message differences 85
14.3.2. Locale differences 85
14.3.3. Date and time differences 86
14.3.4. Floating-point differences 86
14.3.5. Polygon differences 86
14.3.6. Row ordering differences 87
14.3.7. The “random” test 87

14.4. Platform-specific comparison files 87
A. Release Notes 89
A.1. Release 7.2.8 89
A.1.1. Migration to version 7.2.8 89
A.1.2. Changes 89
A.2. Release 7.2.7 89
A.2.1. Migration to version 7.2.7 89
A.2.2. Changes 90
A.3. Release 7.2.6 90
A.3.1. Migration to version 7.2.6 90
A.3.2. Changes 90
A.4. Release 7.2.5 91
A.4.1. Migration to version 7.2.5 91
A.4.2. Changes 91
A.5. Release 7.2.4 91
A.5.1. Migration to version 7.2.4 91
A.5.2. Changes 91
A.6. Release 7.2.3 92
A.6.1. Migration to version 7.2.3 92
A.6.2. Changes 92
A.7. Release 7.2.2 92
A.7.1. Migration to version 7.2.2 92
A.7.2. Changes 92
A.8. Release 7.2.1 93
A.8.1. Migration to version 7.2.1 93
A.8.2. Changes 93
A.9. Release 7.2 94
A.9.1. Overview 94
A.9.2. Migration to version 7.2 95
A.9.3. Changes 95
A.10. Release 7.1.3 105
A.10.1. Migration to version 7.1.3 105
A.10.2. Changes 105
A.11. Release 7.1.2 105
A.11.1. Migration to version 7.1.2 105
A.11.2. Changes 105
A.12. Release 7.1.1 106
A.12.1. Migration to version 7.1.1 106
A.12.2. Changes 106
A.13. Release 7.1 106
A.13.1. Migration to version 7.1 107
A.13.2. Changes 107
A.14. Release 7.0.3 111
A.14.1. Migration to version 7.0.3 111
A.14.2. Changes 111
A.15. Release 7.0.2 112
A.15.1. Migration to version 7.0.2 112
A.15.2. Changes 112
A.16. Release 7.0.1 112
A.16.1. Migration to version 7.0.1 112
A.16.2. Changes 112
A.17. Release 7.0 113
A.17.1. Migration to version 7.0 114
A.17.2. Changes 114

A.18. Release 6.5.3 121
A.18.1. Migration to version 6.5.3 121
A.18.2. Changes 121
A.19. Release 6.5.2 121
A.19.1. Migration to version 6.5.2 121
A.19.2. Changes 121
A.20. Release 6.5.1 122
A.20.1. Migration to version 6.5.1 122
A.20.2. Changes 122
A.21. Release 6.5 123
A.21.1. Migration to version 6.5 124
A.21.2. Changes 124
A.22. Release 6.4.2 128
A.22.1. Migration to version 6.4.2 128
A.22.2. Changes 128
A.23. Release 6.4.1 128
A.23.1. Migration to version 6.4.1 128
A.23.2. Changes 128
A.24. Release 6.4 129
A.24.1. Migration to version 6.4 130
A.24.2. Changes 130
A.25. Release 6.3.2 134
A.25.1. Changes 134
A.26. Release 6.3.1 135
A.26.1. Changes 135
A.27. Release 6.3 136
A.27.1. Migration to version 6.3 137
A.27.2. Changes 137
A.28. Release 6.2.1 141
A.28.1. Migration from version 6.2 to version 6.2.1 141
A.28.2. Changes 141
A.29. Release 6.2 142
A.29.1. Migration from version 6.1 to version 6.2 142
A.29.2. Migration from version 1.*x* to version 6.2 142
A.29.3. Changes 142
A.30. Release 6.1.1 144
A.30.1. Migration from version 6.1 to version 6.1.1 145
A.30.2. Changes 145
A.31. Release 6.1 145
A.31.1. Migration to version 6.1 146
A.31.2. Changes 146
A.32. Release 6.0 148
A.32.1. Migration from version 1.09 to version 6.0 148
A.32.2. Migration from pre-1.09 to version 6.0 148
A.32.3. Changes 148
A.33. Release 1.09 150
A.34. Release 1.02 150
A.34.1. Migration from version 1.02 to version 1.02.1 150
A.34.2. Dump/Reload Procedure 151
A.34.3. Changes 151
A.35. Release 1.01 152
A.35.1. Migration from version 1.0 to version 1.01 152
A.35.2. Changes 154
A.36. Release 1.0 155

A.36.1. Changes 155
A.37. Postgres95 Release 0.03 156
A.37.1. Changes 156
A.38. Postgres95 Release 0.02 158
A.38.1. Changes 158
A.39. Postgres95 Release 0.01 159

List of Tables

3.1. Short option key 30
3.2. System V IPC parameters 31
5.1. Encodings 50
5.2. Communication Encodings 51
10.1. Standard Statistics Views 75
10.2. Statistics Access Functions 76

List of Examples

4.1. An example `pg_hba.conf` file 41
4.2. An example `pg_ident.conf` file 45

Chapter 1. Installation Instructions

1.1. Short Version

```
./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
```

The long version is the rest of this chapter.

1.2. Requirements

In general, a modern Unix-compatible platform should be able to run PostgreSQL. The platforms that had received specific testing at the time of release are listed in Section 1.7, “Supported Platforms” below. In the `doc` subdirectory of the distribution there are several platform-specific FAQ documents you might wish to consult if you are having trouble.

The following prerequisites exist for building PostgreSQL:

- GNU make is required; other make programs will *not* work. GNU make is often installed under the name `gmake`; this document will always refer to it by that name. (On some systems GNU make is the default tool with the name `make`.) To test for GNU make enter

  `gmake --version`

  It is recommended to use version 3.76.1 or later.

- You need an ISO/ANSI C compiler. Recent versions of GCC are recommendable, but PostgreSQL is known to build with a wide variety of compilers from different vendors.

- gzip is needed to unpack the distribution in the first place. If you are reading this, you probably already got past that hurdle.

- The GNU Readline library (for comfortable line editing and command history retrieval) will automatically be used if found. You might wish to install it before proceeding, but it is not essential. (On NetBSD, the `libedit` library is readline-compatible and is used if `libreadline` is not found.)

- GNU Flex and Bison are needed to build from scratch, but they are *not* required when building from a released source package because pre-generated output files are included in released packages. You will need these programs only when building from a CVS tree or if you changed the actual scanner and parser definition files. If you need them, be sure to get Flex 2.5.4 or later and Bison 1.28 or later. Other yacc programs can sometimes be used, but doing so requires extra effort and is not recommended. Other lex programs will definitely not work.

- To build on Windows NT or Windows 2000 you need the Cygwin and cygipc packages. See the file `doc/FAQ_MSWIN` for details.

If you need to get a GNU package, you can find it at your local GNU mirror site (see http://www.gnu.org/order/ftp.html for a list) or at ftp://ftp.gnu.org/gnu/.

Also check that you have sufficient disk space. You will need about 30 MB for the source tree during compilation and about 10 MB for the installation directory. An empty database cluster takes about 20 MB, databases take about five times the amount of space that a flat text file with the same data would take. If you are going to run the regression tests you will temporarily need an extra 20 MB. Use the `df` command to check for disk space.

1.3. Getting The Source

The PostgreSQL 7.2.8 sources can be obtained by anonymous FTP from ftp://ftp.postgresql.org/pub/postgresql-7.2.8.tar.gz. Use a mirror if possible. After you have obtained the file, unpack it:

```
gunzip postgresql-7.2.8.tar.gz
tar xf postgresql-7.2.8.tar
```

This will create a directory `postgresql-7.2.8` under the current directory with the PostgreSQL sources. Change into that directory for the rest of the installation procedure.

1.4. If You Are Upgrading

The internal data storage format changes with new releases of PostgreSQL. Therefore, if you are upgrading an existing installation that does not have a version number "7.2.x", you must back up and restore your data as shown here. These instructions assume that your existing installation is under the `/usr/local/pgsql` directory, and that the data area is in `/usr/local/pgsql/data`. Substitute your paths appropriately.

1. Make sure that your database is not updated during or after the backup. This does not affect the integrity of the backup, but the changed data would of course not be included. If necessary, edit the permissions in the file `/usr/local/pgsql/data/pg_hba.conf` (or equivalent) to disallow access from everyone except you.

2. To dump your database installation, type:

   ```
   pg_dumpall > outputfile
   ```

   If you need to preserve OIDs (such as when using them as foreign keys), then use the `-o` option when running `pg_dumpall`.

   `pg_dumpall` does not save large objects. Check Section 9.1.4, "Caveats" if you need to do this.

   Make sure that you use the `pg_dumpall` command from the version you are currently running. 7.2.8's `pg_dumpall` should not be used on older databases.

3. If you are installing the new version at the same location as the old one then shut down the old server, at the latest before you install the new files:

   ```
   kill -INT `cat /usr/local/pgsql/data/postmaster.pid`
   ```

   Versions prior to 7.0 do not have this `postmaster.pid` file. If you are using such a version you must find out the process id of the server yourself, for example by typing **`ps ax | grep postmaster`**, and supply it to the `kill` command.

On systems that have PostgreSQL started at boot time, there is probably a start-up file that will accomplish the same thing. For example, on a Red Hat Linux system one might find that

```
/etc/rc.d/init.d/postgresql stop
```

works. Another possibility is **pg_ctl stop**.

4. If you are installing in the same place as the old version then it is also a good idea to move the old installation out of the way, in case you have trouble and need to revert to it. Use a command like this:

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

After you have installed PostgreSQL 7.2.8, create a new database directory and start the new server. Remember that you must execute these commands while logged in to the special database user account (which you already have if you are upgrading).

```
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
```

Finally, restore your data with

```
/usr/local/pgsql/bin/psql -d template1 -f outputfile
```

using the *new* psql.

You can also install the new version in parallel with the old one to decrease the downtime. These topics are discussed at length in Section 9.3, “Migration between releases”, which you are encouraged to read in any case.

1.5. Installation Procedure

1. **Configuration**

   The first step of the installation procedure is to configure the source tree for your system and choose the options you would like. This is done by running the `configure` script. For a default installation simply enter

   ```
   ./configure
   ```

   This script will run a number of tests to guess values for various system dependent variables and detect some quirks of your operating system, and finally will create several files in the build tree to record what it found.

   The default configuration will build the server and utilities, as well as all client applications and interfaces that require only a C compiler. All files will be installed under `/usr/local/pgsql` by default.

   You can customize the build and installation process by supplying one or more of the following command line options to `configure`:

   `--prefix=`*`PREFIX`*

   Install all files under the directory *`PREFIX`* instead of `/usr/local/pgsql`. The actual files will be installed into various subdirectories; no files will ever be installed directly into the *`PREFIX`* directory.

If you have special needs, you can also customize the individual subdirectories with the following options.

`--exec-prefix=`*`EXEC-PREFIX`*

You can install architecture-dependent files under a different prefix, *`EXEC-PREFIX`*, than what *`PREFIX`* was set to. This can be useful to share architecture-independent files between hosts. If you omit this, then *`EXEC-PREFIX`* is set equal to *`PREFIX`* and both architecture-dependent and independent files will be installed under the same tree, which is probably what you want.

`--bindir=`*`DIRECTORY`*

Specifies the directory for executable programs. The default is *`EXEC-PREFIX`*`/bin`, which normally means `/usr/local/pgsql/bin`.

`--datadir=`*`DIRECTORY`*

Sets the directory for read-only data files used by the installed programs. The default is *`PREFIX`*`/share`. Note that this has nothing to do with where your database files will be placed.

`--sysconfdir=`*`DIRECTORY`*

The directory for various configuration files, *`PREFIX`*`/etc` by default.

`--libdir=`*`DIRECTORY`*

The location to install libraries and dynamically loadable modules. The default is *`EXEC-PREFIX`*`/lib`.

`--includedir=`*`DIRECTORY`*

The directory for installing C and C++ header files. The default is *`PREFIX`*`/include`.

`--docdir=`*`DIRECTORY`*

Documentation files, except “man” pages, will be installed into this directory. The default is *`PREFIX`*`/doc`.

`--mandir=`*`DIRECTORY`*

The man pages that come with PostgreSQL will be installed under this directory, in their respective `man`*`x`* subdirectories. The default is *`PREFIX`*`/man`.

Note

Care has been taken to make it possible to install PostgreSQL into shared installation locations (such as `/usr/local/include`) without interfering with the namespace of the rest of the system. First, the string “`/postgresql`” is automatically appended to `datadir`, `sysconfdir`, and `docdir`, unless the fully expanded directory name already contains the string “`postgres`” or “`pgsql`”. For example, if you choose `/usr/local` as prefix, the documentation will be installed in `/usr/local/doc/postgresql`, but if the prefix is `/opt/postgres`, then it will be in `/opt/postgres/doc`. Second, the installation layout of the C and C++ header files has been reorganized in the 7.2 release. The public header files of the client interfaces are installed into `includedir` and are namespace-clean. The internal header files and the server header files are installed into

private directories under `includedir`. See the *Programmer's Guide* for information about how to get at the header files for each interface. Finally, a private subdirectory will also be created, if appropriate, under `libdir` for dynamically loadable modules.

`--with-includes=`*`DIRECTORIES`*

`DIRECTORIES` is a colon-separated list of directories that will be added to the list the compiler searches for header files. If you have optional packages (such as GNU Readline) installed in a non-standard location, you have to use this option and probably also the corresponding `--with-libraries` option.

Example: `--with-includes=/opt/gnu/include:/usr/sup/include`.

`--with-libraries=`*`DIRECTORIES`*

`DIRECTORIES` is a colon-separated list of directories to search for libraries. You will probably have to use this option (and the corresponding `--with-includes` option) if you have packages installed in non-standard locations.

Example: `--with-libraries=/opt/gnu/lib:/usr/sup/lib`.

`--enable-locale`

Enables locale support. There is a performance penalty associated with locale support, but if you are not in an English-speaking environment you will most likely need this.

`--enable-recode`

Enables single-byte character set recode support. See Section 5.3, “Single-byte character set recoding” about this feature.

`--enable-multibyte`

Allows the use of multibyte character encodings (including Unicode) and character set encoding conversion. Read Section 5.2, “Multibyte Support” for details.

Note that some interfaces (such as Tcl or Java) expect all character strings to be in Unicode, so this option will be required to correctly support these interfaces.

`--enable-nls[=`*`LANGUAGES`*`]`

Enables Native Language Support (NLS), that is, the ability to display a program's messages in a language other than English. *`LANGUAGES`* is a space separated list of codes of the languages that you want supported, for example `--enable-nls='de fr'`. (The intersection between your list and the set of actually provided translations will be computed automatically.) If you do not specify a list, then all available translations are installed.

To use this option, you will need an implementation of the gettext API. Some operating systems have this built-in (e.g., Linux, NetBSD, Solaris), for other systems you can download an add-on package from here: http://www.postgresql.org/~petere/gettext.html. If you are using the gettext implementation in the GNU C library then you will additionally need the GNU gettext package for some utility programs. For any of the other implementations you will not need it.

`--with-pgport=`*`NUMBER`*

Set *`NUMBER`* as the default port number for server and clients. The default is 5432. The port can always be changed later on, but if you specify it here then both server and clients will have the same default compiled in, which can be very convenient. Usually the only good reason to select a non-default value is if you intend to run multiple PostgreSQL servers on the same machine.

`--with-CXX`

Build the C++ interface library.

`--with-perl`

Build the Perl interface module. The Perl interface will be installed at the usual place for Perl modules (typically under `/usr/lib/perl`), so you must have root access to perform the installation step (see Step 4). You need to have Perl 5 installed to use this option.

`--with-python`

Build the Python interface module. You need to have root access to be able to install the Python module at its default place (`/usr/lib/python`*`x.y`*). To be able to use this option, you must have Python installed and your system needs to support shared libraries. If you instead want to build a new complete interpreter binary, you will have to do it manually.

`--with-tcl`

Builds components that require Tcl/Tk, which are libpgtcl, pgtclsh, pgtksh, PgAccess, and PL/Tcl. But see below about `--without-tk`.

`--without-tk`

If you specify `--with-tcl` and this option, then programs that require Tk (pgtksh and PgAccess) will be excluded.

`--with-tclconfig=`*`DIRECTORY`*
`--with-tkconfig=`*`DIRECTORY`*

Tcl/Tk installs the files `tclConfig.sh` and `tkConfig.sh`, which contain configuration information needed to build modules interfacing to Tcl or Tk. These files are normally found automatically at their well-known locations, but if you want to use a different version of Tcl or Tk you can specify the directory in which to find them.

`--enable-odbc`

Build the ODBC driver. By default, the driver will be independent of a driver manager. To work better with a driver manager already installed on your system, use one of the following options in addition to this one. More information can be found in the *Programmer's Guide*.

`--with-iodbc`

Build the ODBC driver for use with iODBC.

`--with-unixodbc`

Build the ODBC driver for use with unixODBC.

`--with-odbcinst=`*`DIRECTORY`*

Specifies the directory where the ODBC driver will expect its `odbcinst.ini` configuration file. The default is `/usr/local/pgsql/etc` or whatever you specified as `--sysconfdir`. It should be arranged that the driver reads the same file as the driver manager.

If either the option `--with-iodbc` or the option `--with-unixodbc` is used, this option will be ignored because in that case the driver manager handles the location of the configuration file.

`--with-java`

Build the JDBC driver and associated Java packages. This option requires Ant to be installed (as well as a JDK, of course). Refer to the JDBC driver documentation in the *Programmer's Guide* for more information.

`--with-krb4[=`*`DIRECTORY`*`]`
`--with-krb5[=`*`DIRECTORY`*`]`

Build with support for Kerberos authentication. You can use either Kerberos version 4 or 5, but not both. The *`DIRECTORY`* argument specifies the root directory of the Kerberos installation; `/usr/athena` is assumed as default. If the relevant header files and libraries are not under a common parent directory, then you must use the `--with-includes` and `--with-libraries` options in addition to this option. If, on the other hand, the required files are in a location that is searched by default (e.g., `/usr/lib`), then you can leave off the argument.

`configure` will check for the required header files and libraries to make sure that your Kerberos installation is sufficient before proceeding.

`--with-krb-srvnam=`*`NAME`*

The name of the Kerberos service principal. `postgres` is the default. There's probably no reason to change this.

`--with-openssl[=`*`DIRECTORY`*`]`

Build with support for SSL (encrypted) connections. This requires the OpenSSL package to be installed. The *`DIRECTORY`* argument specifies the root directory of the OpenSSL installation; the default is `/usr/local/ssl`.

`configure` will check for the required header files and libraries to make sure that your OpenSSL installation is sufficient before proceeding.

`--with-pam`

Build with PAM (Pluggable Authentication Modules) support.

`--enable-syslog`

Enables the PostgreSQL server to use the syslog logging facility. (Using this option does not mean that you must log with syslog or even that it will be done by default, it simply makes it possible to turn that option on at run time.)

`--enable-debug`

Compiles all programs and libraries with debugging symbols. This means that you can run the programs through a debugger to analyze problems. This enlarges the size of the

installed executables considerably, and on non-GCC compilers it usually also disables compiler optimization, causing slowdowns. However, having the symbols available is extremely helpful for dealing with any problems that may arise. Currently, this option is recommended for production installations only if you use GCC. But you should always have it on if you are doing development work or running a beta version.

`--enable-cassert`

Enables *assertion* checks in the server, which test for many "can't happen" conditions. This is invaluable for code development purposes, but the tests slow things down a little. Also, having the tests turned on won't necessarily enhance the stability of your server! The assertion checks are not categorized for severity, and so what might be a relatively harmless bug will still lead to server restarts if it triggers an assertion failure. Currently, this option is not recommended for production use, but you should have it on for development work or when running a beta version.

`--enable-depend`

Enables automatic dependency tracking. With this option, the makefiles are set up so that all affected object files will be rebuilt when any header file is changed. This is useful if you are doing development work, but is just wasted overhead if you intend only to compile once and install. At present, this option will work only if you use GCC.

If you prefer a C or C++ compiler different from the one `configure` picks then you can set the environment variables `CC` or `CXX`, respectively, to the program of your choice. Similarly, you can override the default compiler flags with the `CFLAGS` and `CXXFLAGS` variables. For example:

```
env CC=/opt/bin/gcc CFLAGS='-O2 -pipe' ./configure
```

2. **Build**

To start the build, type

```
gmake
```

(Remember to use GNU make.) The build may take anywhere from 5 minutes to half an hour depending on your hardware. The last line displayed should be

```
All of PostgreSQL is successfully made. Ready to install.
```

3. **Regression Tests**

If you want to test the newly built server before you install it, you can run the regression tests at this point. The regression tests are a test suite to verify that PostgreSQL runs on your machine in the way the developers expected it to. Type

```
gmake check
```

(This won't work as root; do it as an unprivileged user.) It is possible that some tests fail, due to differences in error message wording or floating point results. Chapter 14, *Regression Tests* contains detailed information about interpreting the test results. You can repeat this test at any later time by issuing the same command.

4. **Installing The Files**

   ## Note

   If you are upgrading an existing system and are going to install the new files over the old ones, then you should have backed up your data and shut down the old server by now, as explained in Section 1.4, “If You Are Upgrading” above.

   To install PostgreSQL enter

   ```
   gmake install
   ```

   This will install files into the directories that were specified in Step 1. Make sure that you have appropriate permissions to write into that area. Normally you need to do this step as root. Alternatively, you could create the target directories in advance and arrange for appropriate permissions to be granted.

   If you built the Perl or Python interfaces and you were not the root user when you executed the above command then that part of the installation probably failed. In that case you should become the root user and then do

   ```
   gmake -C src/interfaces/perl5 install
   gmake -C src/interfaces/python install
   ```

   If you do not have superuser access you are on your own: you can still take the required files and place them in other directories where Perl or Python can find them, but how to do that is left as an exercise.

   The standard installation provides only the header files needed for client application development. If you plan to do any server-side program development (such as custom functions or data types written in C), then you may want to install the entire PostgreSQL include tree into your target include directory. To do that, enter

   ```
   gmake install-all-headers
   ```

   This adds a megabyte or two to the installation footprint, and is only useful if you don't plan to keep the whole source tree around for reference. (If you do, you can just use the source's include directory when building server-side software.)

   Client-only installation: If you want to install only the client applications and interface libraries, then you can use these commands:

   ```
   gmake -C src/bin install
   gmake -C src/include install
   gmake -C src/interfaces install
   gmake -C doc install
   ```

   To undo the installation use the command `gmake uninstall`. However, this will not remove any created directories.

After the installation you can make room by removing the built files from the source tree with the `gmake clean` command. This will preserve the files made by the configure program, so that you can rebuild everything with `gmake` later on. To reset the source tree to the state in which it was distributed, use `gmake distclean`. If you are going to build for several platforms from the same source tree you must do this and re-configure for each build.

If you perform a build and then discover that your configure options were wrong, or if you change anything that configure investigates (for example, you install GNU Readline), then it's a good idea to do `gmake distclean` before reconfiguring and rebuilding. Without this, your changes in configuration choices may not propagate everywhere they need to.

1.6. Post-Installation Setup

1.6.1. Shared Libraries

On some systems that have shared libraries (which most systems do) you need to tell your system how to find the newly installed shared libraries. The systems on which this is *not* necessary include BSD/OS, FreeBSD, HP-UX, IRIX, Linux, NetBSD, OpenBSD, Tru64 UNIX (formerly Digital UNIX), and Solaris.

The method to set the shared library search path varies between platforms, but the most widely usable method is to set the environment variable `LD_LIBRARY_PATH` like so: In Bourne shells (`sh`, `ksh`, `bash`, `zsh`)

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

or in `csh` or `tcsh`

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

Replace `/usr/local/pgsql/lib` with whatever you set `--libdir` to in Step 1. You should put these commands into a shell start-up file such as `/etc/profile` or `~/.bash_profile`. Some good information about the caveats associated with this method can be found at http://www.visi.com/~barr/ldpath.html.

On some systems it might be preferable to set the environment variable `LD_RUN_PATH` *before* building.

If in doubt, refer to the manual pages of your system (perhaps `ld.so` or `rld`). If you later on get a message like

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or
 directory
```

then this step was necessary. Simply take care of it then.

If you are on BSD/OS, Linux, or SunOS 4 and you have root access you can run

```
/sbin/ldconfig /usr/local/pgsql/lib
```

(or equivalent directory) after installation to enable the run-time linker to find the shared libraries faster. Refer to the manual page of `ldconfig` for more information. On FreeBSD, NetBSD, and OpenBSD the command is

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

instead. Other systems are not known to have an equivalent command.

1.6.2. Environment Variables

If you installed into /usr/local/pgsql or some other location that is not searched for programs by default, you need to add /usr/local/pgsql/bin (or whatever you set --bindir to in Step 1) into your PATH. To do this, add the following to your shell start-up file, such as ~/.bash_profile (or /etc/profile, if you want it to affect every user):

```
PATH=/usr/local/pgsql/bin:$PATH
```

If you are using csh or tcsh, then use this command:

```
set path = ( /usr/local/pgsql/bin $path )
```

To enable your system to find the man documentation, you need to add a line like the following to a shell start-up file:

```
MANPATH=/usr/local/pgsql/man:$MANPATH
```

The environment variables PGHOST and PGPORT specify to client applications the host and port of the database server, overriding the compiled-in defaults. If you are going to run client applications remotely then it is convenient if every user that plans to use the database sets PGHOST. This is not required, however: the settings can be communicated via command line options to most client programs.

1.7. Supported Platforms

PostgreSQL has been verified by the developer community to work on the platforms listed below. A supported platform generally means that PostgreSQL builds and installs according to these instructions and that the regression tests pass.

Note

If you are having problems with the installation on a supported platform, please write to <pgsql-bugs@postgresql.org> or <pgsql-ports@postgresql.org>, not to the people listed here.

OS	Processor	Version	Reported	Remarks
AIX	RS6000	7.2	2001-12-19, Andreas Zeugswetter (<ZeugswetterA@spardat.at>), Tatsuo Ishii (<t-ishii@sra.co.jp>)	see also doc/FAQ_AIX
BeOS	x86	7.2	2001-11-29, Cyril Velter (<cyril.velter@libertysurf.fr>)	5.0.4
BSD/OS	x86	7.2	2001-11-27, Bruce Momjian (<pgman@candle.pha.pa.us>)	4.2
FreeBSD	Alpha	7.2	2001-12-18, Chris Kings-Lynne (<chriskl@familyhealth.com.au>)	
FreeBSD	x86	7.2	2001-11-14, Chris Kings-Lynne (<chriskl@familyhealth.com.au>)	

OS	Processor	Version	Reported	Remarks
HP-UX	PA-RISC	7.2	2001-11-29, Joseph Conway (<Joseph.Conway@m.HPQ.com>), Tom Lane (<tgl@sss.pgh.pa.us>)	11.00 and 10.20; see also doc/FAQ_HPUX
IRIX	MIPS	7.2	2001-11-28, Luis Amigo (<lamigo@atc.unican.es>)	6.5.13, MIPSPro 7.30
Linux	Alpha	7.2	2001-11-16, Tom Lane (<tgl@sss.pgh.pa.us>)	2.2.18; tested at SourceForge
Linux	armv4l	7.2	2001-12-10, Mark Knox (<segfault@hardline.org>)	2.2.x
Linux	MIPS	7.2	2001-11-15, Hisao Shibuya (<shibuya@alpha.or.jp>)	2.0.x; Cobalt Qube2
Linux	PlayStation 2	7.2	2001-12-12, Permaine Cheung <pcheung@redhat.com>)	#undef HAS_TEST_AND_SET, slock_t
Linux	PPC74xx	7.2	2001-11-16, Tom Lane (<tgl@sss.pgh.pa.us>)	2.2.18; Apple G3
Linux	S/390	7.2	2001-12-12, Permaine Cheung <pcheung@redhat.com>)	
Linux	Sparc	7.2	2001-11-28, Doug McNaught (<doug@wireboard.com>)	2.2.19
Linux	x86	7.2	2001-11-15, Thomas Lockhart (<lockhart@fourpalms.org>)	2.0.x, 2.2.x, 2.4.x
MacOS X	PPC	7.2	2001-11-28, Gavin Sherry (<swm@linuxworld.com.au>)	10.1.x
NetBSD	Alpha	7.2	2001-11-20, Thomas Thai (<tom@minnesota.com>)	1.5W
NetBSD	arm32	7.1	2001-03-21, Patrick Welche (<prlw1@cam.ac.uk>)	1.5E
NetBSD	m68k	7.0	2000-04-10, Henry B. Hotz (<hotz@jpl.nasa.gov>)	Mac 8xx
NetBSD	PPC	7.2	2001-11-28, Bill Studenmund (<wrstuden@netbsd.org>)	1.5

OS	Processor	Version	Reported	Remarks
NetBSD	Sparc	7.2	2001-12-03, Matthew Green (<mrg@eterna.com.au>)	32- and 64-bit builds
NetBSD	VAX	7.1	2001-03-30, Tom I. Helbekkmo (<tih@kpnQwest.no>)	1.5
NetBSD	x86	7.2	2001-11-28, Bill Studenmund (<wrstuden@netbsd.org>)	1.5
OpenBSD	Sparc	7.2	2001-11-27, Brandon Palmer (<bpalmer@crimelabs.net>)	3.0
OpenBSD	x86	7.2	2001-11-26, Brandon Palmer (<bpalmer@crimelabs.net>)	3.0
Open UNIX	x86	7.2	2001-11-28, OU-8 Larry Rosenman (<ler@lerctr.org>), UW-7 Olivier Prenant (<ohp@pyrenet.fr>)	see also doc/FAQ_SCO
QNX 4 RTOS	x86	7.2	2001-12-10, Bernd Tegge (<tegge@repas-aeg.de>)	4.25; see also doc/FAQ_QNX4
Solaris	Sparc	7.2	2001-11-12, Andrew Sullivan (<andrew@libertyRMS.com>)	2.6-8; see also doc/FAQ_Solaris
Solaris	x86	7.2	2001-11-28, Martin Renters (<martin@dataf ax.com>)	2.8; see also doc/FAQ_Solaris
SunOS 4	Sparc	7.2	2001-12-04, Tatsuo Ishii (<t-ishii@sra.co.jp>)	
Tru64 UNIX	Alpha	7.2	2001-11-26, Alessio Bragadini (<alessio@albourne.com>), Bernd Tegge (<tegge@repas-aeg.de>)	5.0; 4.0g with cc and gcc
Windows	x86	7.2	2001-12-13, Dave Page (<dpage@vale-housing.co.uk>), Jason Tishler (<jason@tishler.net>)	with Cygwin; see doc/FAQ_MSWIN
Windows	x86	7.2	2001-12-10, Dave Page (<dpage@vale-housing.co.uk>)	native is client-side only; see Chapter 2, *Installation on Windows*

Unsupported Platforms: The following platforms are either known not to work, or they used to work in a previous release and we did not receive explicit confirmation of a successful test with version 7.2 at the time this list was compiled. We include these here to let you know that these platforms *could* be supported if given some attention.

OS	Processor	Version	Reported	Remarks
DG/UX 5.4R4.11	m88k	6.3	1998-03-01, Brian E Gallew (<geek+@cmu.edu>)	no recent reports
MkLinux DR1	PPC750	7.0	2001-04-03, Tatsuo Ishii (<t-ishii@sra.co.jp>)	7.1 needs OS update?
NeXTSTEP	x86	6.x	1998-03-01, David Wetzel (<dave@turbocat.de>)	bit rot suspected
QNX RTOS v6	x86	7.2	2001-11-20, Igor Kovalenko (<Igor.Kovalenko@motorola.com>)	patches available in archives, but too late for 7.2
SCO OpenServer 5	x86	6.5	1999-05-25, Andrew Merrill (<andrew@compuserve.com>)	7.2.8 should work, but no reports; see also doc/FAQ_SCO
System V R4	m88k	6.2.1	1998-03-01, Doug Winterburn (<dlw@seavme.xroads.com>)	needs new TAS spinlock code
System V R4	MIPS	6.4	1998-10-28, Frank Ridderbusch (<ridderbusch.pad@sni.de>)	no recent reports
Ultrix	MIPS	7.1	2001-03-26	TAS spinlock code not detected
Ultrix	VAX	6.x	1998-03-01	

Chapter 2. Installation on Windows

Build, installation, and use instructions for PostgreSQL client libraries on Windows

Although PostgreSQL is written for Unix-like operating systems, the C client library (libpq) and the interactive terminal (psql) can be compiled natively under Windows. The makefiles included in the source distribution are written for Microsoft Visual C++ and will probably not work with other systems. It should be possible to compile the libraries manually in other cases.

Tip

If you are using Windows 98 or newer you can build and use all of PostgreSQL "the Unix way" if you install the Cygwin toolkit first. In that case see Chapter 1, *Installation Instructions*.

To build everything that you can on Windows, change into the `src` directory and type the command

```
nmake /f win32.mak
```

This assumes that you have Visual C++ in your path.

The following files will be built:

`interfaces\libpq\Release\libpq.dll`

The dynamically linkable frontend library

`interfaces\libpq\Release\libpqdll.lib`

Import library to link your program to `libpq.dll`

`interfaces\libpq\Release\libpq.lib`

Static library version of the frontend library

`bin\psql\Release\psql.exe`

The PostgreSQL interactive terminal

The only file that really needs to be installed is the `libpq.dll` library. This file should in most cases be placed in the `WINNT\SYSTEM32` directory (or in `WINDOWS\SYSTEM` on a Windows 95/98/ME system). If this file is installed using a setup program, it should be installed with version checking using the `VERSIONINFO` resource included in the file, to ensure that a newer version of the library is not overwritten.

If you plan to do development using libpq on this machine, you will have to add the `src\include` and `src\interfaces\libpq` subdirectories of the source tree to the include path in your compilers settings.

To use the libraries, you must add the `libpqdll.lib` file to your project. (In Visual C++, just right-click on the project and choose to add it.)

Chapter 3. Server Runtime Environment

This chapter discusses how to set up and run the database server and the interactions with the operating system.

3.1. The PostgreSQL user account

As with any other server daemon that is connected to the world at large, it is advisable to run PostgreSQL under a separate user account. This user account should only own the data itself that is being managed by the server, and should not be shared with other daemons. (Thus, using the user "nobody" is a bad idea.) It is not advisable to install the executables as owned by this user account because that runs the risk of user-defined functions gone astray or any other exploits compromising the executable programs.

To add a user account to your system, look for a command `useradd` or `adduser`. The user name postgres is often used but by no means required.

3.2. Creating a database cluster

Before you can do anything, you must initialize a database storage area on disk. We call this a *database cluster*. (SQL speaks of a catalog cluster instead.) A database cluster is a collection of databases that will be accessible through a single instance of a running database server. After initialization, a database cluster will contain one database named `template1`. As the name suggests, this will be used as a template for subsequently created databases; it should not be used for actual work.

In file system terms, a database cluster will be a single directory under which all data will be stored. We call this the *data directory* or *data area*. It is completely up to you where you choose to store your data. There is no default, although locations such as `/usr/local/pgsql/data` or `/var/lib/pgsql/data` are popular. To initialize a database cluster, use the command `initdb`, which is installed with PostgreSQL. The desired file system location of your database system is indicated by the `-D` option, for example

```
$ initdb -D /usr/local/pgsql/data
```

Note that you must execute this command while being logged into the PostgreSQL user account, which is described in the previous section.

Tip

As an alternative to the `-D` option, you can set the environment variable `PGDATA`.

`initdb` will attempt to create the directory you specify if it does not already exist. It is likely that it won't have the permission to do so (if you followed our advice and created an unprivileged account). In that case you should create the directory yourself (as root) and transfer ownership of it to the PostgreSQL user account. Here is how this might work:

```
root# mkdir /usr/local/pgsql/data
root# chown postgres /usr/local/pgsql/data
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

`initdb` will refuse to run if the data directory looks like it belongs to an already initialized installation.

Because the data directory contains all the data stored in the database, it is essential that it be well secured from unauthorized access. `initdb` therefore revokes access permissions from everyone but the PostgreSQL user account.

However, while the directory contents are secure, the default `pg_hba.conf` authentication method of `trust` allows any local user to connect to the database and even become the database superuser. If you don't trust other local users, we recommend you use `initdb`'s option `-W` or `--pwprompt` to assign a password to the database superuser. After `initdb`, modify `pg_hba.conf` to use `md5` or `password`, instead of `trust`, authentication *before* you start the server for the first time. (Other, possibly more convenient approaches include using `ident` authentication or file system permissions to restrict connections. See Chapter 4, *Client Authentication* for more information.)

One surprise you might encounter while running `initdb` is a notice similar to this one:

```
NOTICE:  Initializing database with en_US collation order.
        This locale setting will prevent use of index optimization for
        LIKE and regexp searches.  If you are concerned about speed of
        such queries, you may wish to set LC_COLLATE to "C" and
        re-initdb.  For more information see the Administrator's
 Guide.
```

This notice is intended to warn you that the currently selected locale will cause indexes to be sorted in an order that prevents them from being used for LIKE and regular-expression searches. If you need good performance of such searches, you should set your current locale to `C` and re-run `initdb`. On most systems, setting the current locale is done by changing the value of the environment variable `LC_ALL` or `LANG`. The sort order used within a particular database cluster is set by `initdb` and cannot be changed later, short of dumping all data, rerunning `initdb`, and reloading the data. So it's important to make this choice correctly now.

3.3. Starting the database server

Before anyone can access the database you must start the database server. The database server is called *postmaster*. The postmaster must know where to find the data it is supposed to work on. This is done with the `-D` option. Thus, the simplest way to start the server is, for example,

```
$ postmaster -D /usr/local/pgsql/data
```

which will leave the server running in the foreground. This must again be done while logged into the PostgreSQL user account. Without a `-D`, the server will try to use the data directory in the environment variable `PGDATA`; if neither of these works it will fail.

To start the postmaster in the background, use the usual shell syntax:

```
$ postmaster -D /usr/local/pgsql/data > logfile 2>&1 &
```

It is an extremely good idea to keep the server's stdout and stderr output around somewhere, as suggested here. It will help both for auditing purposes and to diagnose problems. (See Section 8.3, “Log File Maintenance” for a more thorough discussion of log file handling.)

The postmaster also takes a number of other command line options. For more information see the reference page and Section 3.4, “Run-time configuration” below. In particular, in order for the server to accept TCP/IP connections (rather than just Unix domain socket ones), you must also specify the `-i` option.

This shell syntax can get tedious quickly. Therefore the shell script wrapper pg_ctl is provided that encapsulates some of the tasks. E.g.,

```
pg_ctl start -l logfile
```

will start the server in the background and put the output into the named log file. The `-D` option has the same meaning as when invoking postmaster directly. pg_ctl also implements a symmetric "stop" operation.

Normally, you will want to start the database server when the computer boots up. This is not required; the PostgreSQL server can be run successfully from non-privileged accounts without root intervention.

Different systems have different conventions for starting up daemons at boot time, so you are advised to familiarize yourself with them. Many systems have a file `/etc/rc.local` or `/etc/rc.d/rc.local` which is almost certainly no bad place to put such a command. Whatever you do, the server must be run by the PostgreSQL user account *and not by root* or any other user. Therefore you probably always want to form your command lines along the lines of `su -c '...' postgres`, for example:

```
su -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog' postgres
```

Here are a few more operating system specific suggestions. (Always replace the proper installation directory and the user name you chose.)

- For FreeBSD, take a look at the file `contrib/start-scripts/freebsd` in the PostgreSQL source distribution.
- On OpenBSD, add the following lines to the file `/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/
  log -s' postgres
      echo -n ' postgresql'
  fi
  ```

- On Linux systems either add

  ```
  /usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/
  data
  ```

  to `/etc/rc.d/rc.local` or look into the file `contrib/start-scripts/linux` in the PostgreSQL source distribution to integrate the start and shutdown into the run level system.
- On NetBSD, either use the FreeBSD or Linux start scripts, depending on preference, as an example and place the file at `/usr/local/etc/rc.d/postgresql`.
- On Solaris, create a file called `/etc/init.d/postgresql` to contain the following single line:

  ```
  su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l logfile -D /
  usr/local/pgsql/data"
  ```

  Then, create a symbolic link to it in `/etc/rc3.d` as `S99postgresql`.

While the postmaster is running, its PID is in the file `postmaster.pid` in the data directory. This is used as an interlock against multiple postmasters running in the same data directory, and can also be used for shutting down the postmaster.

3.3.1. Server Start-up Failures

There are several common reasons for the postmaster to fail to start up. Check the postmaster's log file, or start it by hand (without redirecting standard output or standard error) to see what complaint messages appear. Some of the possible error messages are reasonably self-explanatory, but here are some that are not.

```
FATAL: StreamServerPort: bind() failed: Address already in use
        Is another postmaster already running on that port?
```

This usually means just what it suggests: you tried to start a second postmaster on the same port where one is already running. However, if the kernel error message is not `Address already in use` or some variant of that wording, there may be a different problem. For example, trying to start a postmaster on a reserved port number may draw something like

```
$ postmaster -i -p 666
FATAL: StreamServerPort: bind() failed: Permission denied
        Is another postmaster already running on that port?
```

A message like

```
IpcMemoryCreate: shmget(key=5440001, size=83918612, 01600) failed:
 Invalid argument
FATAL 1:  ShmemCreate: cannot create region
```

probably means that your kernel's limit on the size of shared memory areas is smaller than the buffer area that PostgreSQL is trying to create (83918612 bytes in this example). Or it could mean that you don't have System-V-style shared memory support configured into your kernel at all. As a temporary workaround, you can try starting the postmaster with a smaller-than-normal number of buffers (`-B` switch). You will eventually want to reconfigure your kernel to increase the allowed shared memory size, however. You may see this message when trying to start multiple postmasters on the same machine, if their total space requests exceed the kernel limit.

An error like

```
IpcSemaphoreCreate: semget(key=5440026, num=16, 01600) failed: No
 space left on device
```

does *not* mean that you've run out of disk space; it means that your kernel's limit on the number of System V semaphores is smaller than the number PostgreSQL wants to create. As above, you may be able to work around the problem by starting the postmaster with a reduced number of backend processes (`-N` switch), but you'll eventually want to increase the kernel limit.

If you get an “illegal system call” error, then it is likely that shared memory or semaphores are not supported at all in your kernel. In that case your only option is to re-configure the kernel to turn on these features.

Details about configuring System V IPC facilities are given in Section 3.5.1, “Shared Memory and Semaphores”.

3.3.2. Client Connection Problems

Although the possible error conditions on the client side are both virtually infinite and application-dependent, a few of them might be directly related to how the server was started up. Conditions other than those shown below should be documented with the respective client application.

```
psql: could not connect to server: Connection refused
        Is the server running on host server.joe.com and accepting
        TCP/IP connections on port 5432?
```

This is the generic “I couldn't find a server to talk to” failure. It looks like the above when TCP/IP communication is attempted. A common mistake is to forget the `-i` option to allow the postmaster to accept TCP/IP connections.

Alternatively, you'll get this when attempting Unix-socket communication to a local postmaster:

```
psql: could not connect to server: Connection refused
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

The last line is useful in verifying that the client is trying to connect where it is supposed to. If there is in fact no postmaster running there, the kernel error message will typically be either `Connection refused` or `No such file or directory`, as illustrated. (It is particularly important to realize that `Connection refused` in this context does *not* mean that the postmaster got your connection request and rejected it -- that case will produce a different message, as shown in Section 4.3, “Authentication problems”.) Other error messages such as `Connection timed out` may indicate more fundamental problems, like lack of network connectivity.

3.4. Run-time configuration

There are a lot of configuration parameters that affect the behavior of the database system in some way or other. Here we describe how to set them and the following subsections will discuss each of them.

All parameter names are case-insensitive. Every parameter takes a value of one of the four types Boolean, integer, floating point, string as described below. Boolean values are `ON`, `OFF`, `TRUE`, `FALSE`, `YES`, `NO`, `1`, `0` (case-insensitive) or any non-ambiguous prefix of these.

One way to set these options is to edit the file `postgresql.conf` in the data directory. (A default file is installed there.) An example of what this file could look like is:

```
# This is a comment
log_connections = yes
syslog = 2
```

As you see, options are one per line. The equal sign between name and value is optional. Whitespace is insignificant, blank lines are ignored. Hash marks (“#”) introduce comments anywhere.

The configuration file is reread whenever the postmaster receives a SIGHUP signal (which is most easily sent by means of `pg_ctl reload`). The postmaster also propagates this signal to all already-running backend processes, so that existing sessions also get the new default. Alternatively, you can send the signal to only one backend process directly.

A second way to set these configuration parameters is to give them as a command line option to the postmaster, such as

```
postmaster -c log_connections=yes -c syslog=2
```

which would have the same effect as the previous example. Command-line options override any conflicting settings in `postgresql.conf`.

Occasionally it is also useful to give a command line option to one particular backend session only. The environment variable `PGOPTIONS` can be used for this purpose on the client side:

```
env PGOPTIONS='-c geqo=off' psql
```

(This works for any client application, not just psql.) Note that this won't work for options that are necessarily fixed once the server is started, such as the port number.

Finally, some options can be changed in individual SQL sessions with the `SET` command, for example

```
=> SET ENABLE_SEQSCAN TO OFF;
```

See the SQL command language reference for details on the syntax.

3.4.1. Planner and Optimizer Tuning

`CPU_INDEX_TUPLE_COST` (`floating point`)

Sets the query optimizer's estimate of the cost of processing each index tuple during an index scan. This is measured as a fraction of the cost of a sequential page fetch.

`CPU_OPERATOR_COST` (`floating point`)

Sets the optimizer's estimate of the cost of processing each operator in a WHERE clause. This is measured as a fraction of the cost of a sequential page fetch.

`CPU_TUPLE_COST` (`floating point`)

Sets the query optimizer's estimate of the cost of processing each tuple during a query. This is measured as a fraction of the cost of a sequential page fetch.

`EFFECTIVE_CACHE_SIZE` (`floating point`)

Sets the optimizer's assumption about the effective size of the disk cache (that is, the portion of the kernel's disk cache that will be used for PostgreSQL data files). This is measured in disk pages, which are normally 8 kB apiece.

`ENABLE_HASHJOIN` (`boolean`)

Enables or disables the query planner's use of hash-join plan types. The default is on. This is mostly useful to debug the query planner.

`ENABLE_INDEXSCAN` (`boolean`)

Enables or disables the query planner's use of index-scan plan types. The default is on. This is mostly useful to debug the query planner.

`ENABLE_MERGEJOIN` (`boolean`)

Enables or disables the query planner's use of merge-join plan types. The default is on. This is mostly useful to debug the query planner.

`ENABLE_NESTLOOP` (`boolean`)

Enables or disables the query planner's use of nested-loop join plans. It's not possible to suppress nested-loop joins entirely, but turning this variable off discourages the planner from using one if there is any other method available. The default is on. This is mostly useful to debug the query planner.

`ENABLE_SEQSCAN` (`boolean`)

Enables or disables the query planner's use of sequential scan plan types. It's not possible to suppress sequential scans entirely, but turning this variable off discourages the planner from using one if there is any other method available. The default is on. This is mostly useful to debug the query planner.

`ENABLE_SORT` (`boolean`)

Enables or disables the query planner's use of explicit sort steps. It's not possible to suppress explicit sorts entirely, but turning this variable off discourages the planner from using one if there is any other method available. The default is on. This is mostly useful to debug the query planner.

`ENABLE_TIDSCAN` (`boolean`)

Enables or disables the query planner's use of TID scan plan types. The default is on. This is mostly useful to debug the query planner.

`GEQO` (`boolean`)

Enables or disables genetic query optimization, which is an algorithm that attempts to do query planning without exhaustive search. This is on by default. See also the various other `GEQO_` settings.

`GEQO_EFFORT` (`integer`)
`GEQO_GENERATIONS` (`integer`)
`GEQO_POOL_SIZE` (`integer`)
`GEQO_RANDOM_SEED` (`integer`)
`GEQO_SELECTION_BIAS` (`floating point`)

Various tuning parameters for the genetic query optimization algorithm: The pool size is the number of individuals in one population. Valid values are between 128 and 1024. If it is set to 0 (the default) a pool size of 2^(QS+1), where QS is the number of FROM items in the query, is taken. The effort is used to calculate a default for generations. Valid values are between 1 and 80, 40 being the default. Generations specifies the number of iterations in the algorithm. The number must be a positive integer. If 0 is specified then `Effort * Log2(PoolSize)` is used. The run time of the algorithm is roughly proportional to the sum of pool size and generations. The selection bias is the selective pressure within the population. Values can be from 1.50 to 2.00; the latter is the default. The random seed can be set to get reproducible results from the algorithm. If it is set to -1 then the algorithm behaves non-deterministically.

`GEQO_THRESHOLD` (`integer`)

Use genetic query optimization to plan queries with at least this many `FROM` items involved. (Note that a `JOIN` construct counts as only one `FROM` item.) The default is 11. For simpler queries it is usually best to use the deterministic, exhaustive planner. This parameter also controls how hard the optimizer will try to merge subquery `FROM` clauses into the upper query.

`KSQO` (`boolean`)

The *Key Set Query Optimizer* (KSQO) causes the query planner to convert queries whose `WHERE` clause contains many OR'ed AND clauses (such as `WHERE (a=1 AND b=2) OR (a=2 AND b=3) ...`) into a union query. This method can be faster than the default implementation, but it doesn't necessarily give exactly the same results, since `UNION` implicitly adds a `SELECT DISTINCT` clause to eliminate identical output rows. KSQO is commonly used when working with products like Microsoft Access, which tend to generate queries of this form.

The KSQO algorithm used to be absolutely essential for queries with many OR'ed AND clauses, but in PostgreSQL 7.0 and later the standard planner handles these queries fairly successfully. Hence the default is off.

`RANDOM_PAGE_COST` (`floating point`)

Sets the query optimizer's estimate of the cost of a nonsequentially fetched disk page. This is measured as a multiple of the cost of a sequential page fetch.

Note

Unfortunately, there is no well-defined method of determining ideal values for the family of "COST" variables that were just described. You are encouraged to experiment and share your findings.

3.4.2. Logging and Debugging

`DEBUG_ASSERTIONS` (`boolean`)

Turns on various assertion checks. This is a debugging aid. If you are experiencing strange problems or crashes you might want to turn this on, as it might expose programming mistakes. To use this option, the macro `USE_ASSERT_CHECKING` must be defined when PostgreSQL is built (see the configure option `--enable-cassert`). Note that `DEBUG_ASSERTIONS` defaults to on if PostgreSQL has been built this way.

`DEBUG_LEVEL` (`integer`)

The higher this value is set, the more "debugging" output of various sorts is generated in the server log during operation. This option is 0 by default, which means no debugging output. Values up to about 4 currently make sense.

`DEBUG_PRINT_QUERY` (`boolean`)
`DEBUG_PRINT_PARSE` (`boolean`)
`DEBUG_PRINT_REWRITTEN` (`boolean`)
`DEBUG_PRINT_PLAN` (`boolean`)
`DEBUG_PRETTY_PRINT` (`boolean`)

These flags enable various debugging output to be sent to the server log. For each executed query, prints either the query text, the resulting parse tree, the query rewriter output, or the execution plan. `DEBUG_PRETTY_PRINT` indents these displays to produce a more readable but much longer output format. Setting `DEBUG_LEVEL` above zero implicitly turns on some of these flags.

`HOSTNAME_LOOKUP` (`boolean`)

By default, connection logs only show the IP address of the connecting host. If you want it to show the host name you can turn this on, but depending on your host name resolution setup it might impose a non-negligible performance penalty. This option can only be set at server start.

`LOG_CONNECTIONS` (`boolean`)

Prints a line informing about each successful connection in the server log. This is off by default, although it is probably very useful. This option can only be set at server start or in the `postgresql.conf` configuration file.

`LOG_PID` (`boolean`)

Prefixes each server log message with the process ID of the backend process. This is useful to sort out which messages pertain to which connection. The default is off.

`LOG_TIMESTAMP` (`boolean`)

Prefixes each server log message with a time stamp. The default is off.

`SHOW_QUERY_STATS` (`boolean`)
`SHOW_PARSER_STATS` (`boolean`)
`SHOW_PLANNER_STATS` (`boolean`)
`SHOW_EXECUTOR_STATS` (`boolean`)

For each query, write performance statistics of the respective module to the server log. This is a crude profiling instrument.

`SHOW_SOURCE_PORT` (`boolean`)

Shows the outgoing port number of the connecting host in the connection log messages. You could trace back the port number to find out what user initiated the connection. Other than that it's pretty useless and therefore off by default. This option can only be set at server start.

`STATS_COMMAND_STRING` (`boolean`)
`STATS_BLOCK_LEVEL` (`boolean`)
`STATS_ROW_LEVEL` (`boolean`)

These flags determine what information backends send to the statistics collector process: current commands, block-level activity statistics, or row-level activity statistics. All default to off. Enabling statistics collection costs a small amount of time per query, but is invaluable for debugging and performance tuning.

`STATS_RESET_ON_SERVER_START` (`boolean`)

If on, collected statistics are zeroed out whenever the server is restarted. If off, statistics are accumulated across server restarts. The default is on. This option can only be set at server start.

`STATS_START_COLLECTOR` (`boolean`)

Controls whether the server should start the statistics-collection subprocess. This is on by default, but may be turned off if you know you have no interest in collecting statistics. This option can only be set at server start.

`SYSLOG` (`integer`)

PostgreSQL allows the use of syslog for logging. If this option is set to 1, messages go both to syslog and the standard output. A setting of 2 sends output only to syslog. (Some messages will still go to the standard output/error.) The default is 0, which means syslog is off. This option must be set at server start.

To use syslog, the build of PostgreSQL must be configured with the `--enable-syslog` option.

`SYSLOG_FACILITY` (`string`)

This option determines the syslog “facility” to be used when syslog is enabled. You may choose from LOCAL0, LOCAL1, LOCAL2, LOCAL3, LOCAL4, LOCAL5, LOCAL6, LOCAL7; the default is LOCAL0. See also the documentation of your system's syslog.

`SYSLOG_IDENT` (`string`)

If logging to syslog is enabled, this option determines the program name used to identify PostgreSQL messages in syslog log messages. The default is `postgres`.

`TRACE_NOTIFY` (`boolean`)

Generates a great amount of debugging output for the `LISTEN` and `NOTIFY` commands.

3.4.3. General operation

`AUSTRALIAN_TIMEZONES` (`bool`)

If set to true, `CST`, `EST`, and `SAT` are interpreted as Australian time zones rather than as North American Central/Eastern time zones and Saturday. The default is false.

AUTHENTICATION_TIMEOUT (integer)

Maximum time to complete client authentication, in seconds. If a would-be client has not completed the authentication protocol in this much time, the server unceremoniously breaks the connection. This prevents hung clients from occupying a connection indefinitely. This option can only be set at server start or in the postgresql.conf file.

DEADLOCK_TIMEOUT (integer)

This is the amount of time, in milliseconds, to wait on a lock before checking to see if there is a deadlock condition or not. The check for deadlock is relatively slow, so we don't want to run it every time we wait for a lock. We (optimistically?) assume that deadlocks are not common in production applications, and just wait on the lock for awhile before starting to ask questions about whether it can ever get unlocked. Increasing this value reduces the amount of time wasted in needless deadlock checks, but slows down reporting of real deadlock errors. The default is 1000 (i.e., one second), which is probably about the smallest value you would want in practice. On a heavily loaded server you might want to raise it. Ideally the setting should exceed your typical transaction time, so as to improve the odds that the lock will be released before the waiter decides to check for deadlock. This option can only be set at server start.

DEFAULT_TRANSACTION_ISOLATION (string)

Each SQL transaction has an isolation level, which can be either “read committed” or “serializable”. This parameter controls what the isolation level of each new transaction is set to. The default is read committed.

Consult the *PostgreSQL User's Guide* and the command SET TRANSACTION for more information.

DYNAMIC_LIBRARY_PATH (string)

If a dynamically loadable module needs to be opened and the specified name does not have a directory component (i.e., the name does not contain a slash), the system will search this path for the specified file. (The name that is used is the name specified in the CREATE FUNCTION or LOAD command.)

The value for dynamic_library_path has to be a colon-separated list of absolute directory names. If a directory name starts with the special value $libdir, the compiled-in PostgreSQL package library directory, which is where the modules provided by the PostgreSQL distribution are installed, is substituted. (Use pg_config --pkglibdir to print the name of this directory.) An example value:

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/my_project/
lib:$libdir'
```

The default value for this parameter is $libdir. If the value is set to the empty string, the automatic path search is turned off.

This parameter can be changed at run time by superusers, but note that a setting done that way will only persist till the end of the client connection, so this method should be reserved for development purposes. The recommended way to set this parameter is in the postgresql.conf configuration file.

FSYNC (boolean)

If this option is on, the PostgreSQL backend will use the fsync() system call in several places to make sure that updates are physically written to disk and do not hang around in the kernel buffer cache. This increases the chance by a large amount that a database installation will still be usable

after an operating system or hardware crash. (Crashes of the database server itself do *not* affect this consideration.)

However, this operation slows down PostgreSQL, because at all those points it has to block and wait for the operating system to flush the buffers. Without `fsync`, the operating system is allowed to do its best in buffering, sorting, and delaying writes, which can make for a considerable performance increase. However, if the system crashes, the results of the last few committed transactions may be lost in part or whole; in the worst case, unrecoverable data corruption may occur.

This option is the subject of an eternal debate in the PostgreSQL user and developer communities. Some always leave it off, some turn it off only for bulk loads, where there is a clear restart point if something goes wrong, some leave it on just to be on the safe side. Because it is the safe side, on is also the default. If you trust your operating system, your hardware, and your utility company (or better your UPS), you might want to disable `fsync`.

It should be noted that the performance penalty from doing `fsync`s is considerably less in PostgreSQL version 7.1 than it was in prior releases. If you previously suppressed `fsync`s because of performance problems, you may wish to reconsider your choice.

This option can only be set at server start or in the `postgresql.conf` file.

`KRB_SERVER_KEYFILE` (`string`)

Sets the location of the Kerberos server key file. See Section 4.2.3, “Kerberos authentication” for details.

`MAX_CONNECTIONS` (`integer`)

Determines how many concurrent connections the database server will allow. The default is 32 (unless altered while building the server). This parameter can only be set at server start.

`MAX_EXPR_DEPTH` (`integer`)

Sets the maximum expression nesting depth that the parser will accept. The default value is high enough for any normal query, but you can raise it if you need to. (But if you raise it too high, you run the risk of backend crashes due to stack overflow.)

`MAX_FILES_PER_PROCESS` (`integer`)

Sets the maximum number of simultaneously open files in each server subprocess. The default is 1000. The limit actually used by the code is the smaller of this setting and the result of `sysconf(_SC_OPEN_MAX)`. Therefore, on systems where `sysconf` returns a reasonable limit, you don't need to worry about this setting. But on some platforms (notably, most BSD systems), `sysconf` returns a value that is much larger than the system can really support when a large number of processes all try to open that many files. If you find yourself seeing “Too many open files” failures, try reducing this setting. This option can only be set at server start or in the `postgresql.conf` configuration file; if changed in the configuration file, it only affects subsequently-started server subprocesses.

`MAX_FSM_RELATIONS` (`integer`)

Sets the maximum number of relations (tables) for which free space will be tracked in the shared free-space map. The default is 100. This option can only be set at server start.

`MAX_FSM_PAGES` (`integer`)

Sets the maximum number of disk pages for which free space will be tracked in the shared free-space map. The default is 10000. This option can only be set at server start.

`MAX_LOCKS_PER_TRANSACTION` (`integer`)

The shared lock table is sized on the assumption that at most `max_locks_per_transaction` * `max_connections` distinct objects will need to be locked at any one time. The default, 64, has historically proven sufficient, but you might need to raise this value if you have clients that touch many different tables in a single transaction. This option can only be set at server start.

`PASSWORD_ENCRYPTION` (`boolean`)

When a password is specified in `CREATE USER` or `ALTER USER` without writing either ENCRYPTED or UNENCRYPTED, this flag determines whether the password is to be encrypted. The default is off (do not encrypt the password), but this choice may change in a future release.

`PORT` (`integer`)

The TCP port the server listens on; 5432 by default. This option can only be set at server start.

`SHARED_BUFFERS` (`integer`)

Sets the number of shared memory buffers the database server will use. The default is 64. Each buffer is typically 8192 bytes. This option can only be set at server start.

`SILENT_MODE` (`bool`)

Runs postmaster silently. If this option is set, postmaster will automatically run in background and any controlling ttys are disassociated, thus no messages are written to standard output or standard error (same effect as postmaster's -S option). Unless some logging system such as syslog is enabled, using this option is discouraged since it makes it impossible to see error messages.

`SORT_MEM` (`integer`)

Specifies the amount of memory to be used by internal sorts and hashes before switching to temporary disk files. The value is specified in kilobytes, and defaults to 512 kilobytes. Note that for a complex query, several sorts and/or hashes might be running in parallel, and each one will be allowed to use as much memory as this value specifies before it starts to put data into temporary files. And don't forget that each running backend could be doing one or more sorts. So the total memory space needed could be many times the value of `SORT_MEM`.

`SQL_INHERITANCE` (`bool`)

This controls the inheritance semantics, in particular whether subtables are included into the consideration of various commands by default. This was not the case in versions prior to 7.1. If you need the old behavior you can set this variable to off, but in the long run you are encouraged to change your applications to use the `ONLY` keyword to exclude subtables. See the SQL language reference and the *User's Guide* for more information about inheritance.

`SSL` (`boolean`)

Enables SSL connections. Please read Section 3.7, “Secure TCP/IP Connections with SSL” before using this. The default is off.

`TCPIP_SOCKET` (`boolean`)

If this is true, then the server will accept TCP/IP connections. Otherwise only local Unix domain socket connections are accepted. It is off by default. This option can only be set at server start.

TRANSFORM_NULL_EQUALS (boolean)

When turned on, expressions of the form *expr* = NULL (or NULL = *expr*) are treated as *expr* IS NULL, that is, they return true if *expr* evaluates to the NULL value, and false otherwise. The correct behavior of *expr* = NULL is to always return NULL (unknown). Therefore this option defaults to off.

However, filtered forms in Microsoft Access generate queries that appear to use *expr* = NULL to test for NULLs, so if you use that interface to access the database you might want to turn this option on. Since expressions of the form *expr* = NULL always return NULL (using the correct interpretation) they are not very useful and do not appear often in normal applications, so this option does little harm in practice. But new users are frequently confused about the semantics of expressions involving NULL, so we do not turn this option on by default.

Note that this option only affects the literal = operator, not other comparison operators or other expressions that are computationally equivalent to some expression involving the equals operator (such as IN). Thus, this option is not a general fix for bad programming.

Refer to the *User's Guide* for related information.

UNIX_SOCKET_DIRECTORY (string)

Specifies the directory of the Unix-domain socket on which the postmaster is to listen for connections from client applications. The default is normally /tmp, but can be changed at build time.

UNIX_SOCKET_GROUP (string)

Sets the group owner of the Unix domain socket. (The owning user of the socket is always the user that starts the postmaster.) In combination with the option UNIX_SOCKET_PERMISSIONS this can be used as an additional access control mechanism for this socket type. By default this is the empty string, which uses the default group for the current user. This option can only be set at server start.

UNIX_SOCKET_PERMISSIONS (integer)

Sets the access permissions of the Unix domain socket. Unix domain sockets use the usual Unix file system permission set. The option value is expected to be an numeric mode specification in the form accepted by the chmod and umask system calls. (To use the customary octal format the number must start with a 0 (zero).)

The default permissions are 0777, meaning anyone can connect. Reasonable alternatives would be 0770 (only user and group, see also under UNIX_SOCKET_GROUP) and 0700 (only user). (Note that actually for a Unix socket, only write permission matters and there is no point in setting or revoking read or execute permissions.)

This access control mechanism is independent from the one described in Chapter 4, *Client Authentication*.

This option can only be set at server start.

VACUUM_MEM (integer)

Specifies the maximum amount of memory to be used by VACUUM to keep track of to-be-reclaimed tuples. The value is specified in kilobytes, and defaults to 8192 kilobytes. Larger settings may improve the speed of vacuuming large tables that have many deleted tuples.

VIRTUAL_HOST (string)

Specifies the TCP/IP host name or address on which the postmaster is to listen for connections from client applications. Defaults to listening on all configured addresses (including localhost).

3.4.4. WAL

See also Section 11.3, “WAL Configuration” for details on WAL tuning.

`CHECKPOINT_SEGMENTS` (`integer`)

Maximum distance between automatic WAL checkpoints, in log file segments (each segment is normally 16 megabytes). This option can only be set at server start or in the `postgresql.conf` file.

`CHECKPOINT_TIMEOUT` (`integer`)

Maximum time between automatic WAL checkpoints, in seconds. This option can only be set at server start or in the `postgresql.conf` file.

`COMMIT_DELAY` (`integer`)

Time delay between writing a commit record to the WAL buffer and flushing the buffer out to disk, in microseconds. A nonzero delay allows multiple transactions to be committed with only one `fsync` system call, if system load is high enough that additional transactions become ready to commit within the given interval. But the delay is just wasted time if no other transactions become ready to commit. Therefore, the delay is only performed if at least COMMIT_SIBLINGS other transactions are active at the instant that a backend has written its commit record.

`COMMIT_SIBLINGS` (`integer`)

Minimum number of concurrent open transactions to require before performing the `COMMIT_DELAY` delay. A larger value makes it more probable that at least one other transaction will become ready to commit during the delay interval.

`WAL_BUFFERS` (`integer`)

Number of disk-page buffers in shared memory for WAL log. This option can only be set at server start.

`WAL_DEBUG` (`integer`)

If non-zero, turn on WAL-related debugging output on standard error.

`WAL_FILES` (`integer`)

Number of log files that are created in advance at checkpoint time. This option can only be set at server start or in the `postgresql.conf` file.

`WAL_SYNC_METHOD` (`string`)

Method used for forcing WAL updates out to disk. Possible values are `FSYNC` (call `fsync()` at each commit), `FDATASYNC` (call `fdatasync()` at each commit), `OPEN_SYNC` (write WAL files with `open()` option `O_SYNC`), or `OPEN_DATASYNC` (write WAL files with `open()` option `O_DSYNC`). Not all of these choices are available on all platforms. This option can only be set at server start or in the `postgresql.conf` file.

3.4.5. Short options

For convenience there are also single letter option switches available for many parameters. They are described in the following table.

Table 3.1. Short option key

Short option	Equivalent	Remark
`-B` *x*	`shared_buffers` = *x*	
`-d` *x*	`debug_level` = *x*	
`-F`	`fsync = off`	
`-h` *x*	`virtual_host` = *x*	
`-i`	`tcpip_socket = on`	
`-k` *x*	`unix_socket_directory` = *x*	
`-l`	`ssl = on`	
`-N` *x*	`max_connections` = *x*	
`-p` *x*	`port` = *x*	
`-fi`, `-fh`, `-fm`, `-fn`, `-fs`, `-ft`	`enable_indexscan=off`, `enable_hashjoin=off`, `enable_mergejoin=off`, `enable_nestloop=off`, `enable_seqscan=off`, `enable_tidscan=off`	*
`-S` *x*	`sort_mem` = *x*	*
`-s`	`show_query_stats = on`	*
`-tpa`, `-tpl`, `-te`	`show_parser_stats=on`, `show_planner_stats=on`, `show_executor_stats=on`	*

For historical reasons, options marked "*" must be passed to the individual backend process via the `-o` postmaster option, for example,

```
$ postmaster -o '-S 1024 -s'
```

or via `PGOPTIONS` from the client side, as explained above.

3.5. Managing Kernel Resources

A large PostgreSQL installation can quickly hit various operating system resource limits. (On some systems, the factory defaults are so low that you don't even need a really "large" installation.) If you have encountered this kind of problem then keep reading.

3.5.1. Shared Memory and Semaphores

Shared memory and semaphores are collectively referred to as "System V IPC" (together with message queues, which are not relevant for PostgreSQL). Almost all modern operating systems provide these features, but not all of them have them turned on or sufficiently sized by default, especially systems with BSD heritage. (For the QNX and BeOS ports, PostgreSQL provides its own replacement implementation of these facilities.)

The complete lack of these facilities is usually manifested by an Illegal system call error upon postmaster start. In that case there's nothing left to do but to reconfigure your kernel -- PostgreSQL won't work without them.

When PostgreSQL exceeds one of the various hard limits of the IPC resources then the postmaster will refuse to start up and should leave a marginally instructive error message about which problem was encountered and what needs to be done about it. (See also Section 3.3.1, “Server Start-up Failures”.) The relevant kernel parameters are named consistently across different systems; Table 3.2, “System V IPC parameters” gives an overview. The methods to set them, however, vary; suggestions for some platforms are given below. Be warned that it is often necessary to reboot your machine at least, possibly even recompile the kernel, to change these settings.

Table 3.2. System V IPC parameters

Name	Description	Reasonable values
SHMMAX	Maximum size of shared memory segment (bytes)	250kB + 8.2kB * shared_buffers + 14.2kB * max_connections or infinity
SHMMIN	Minimum size of shared memory segment (bytes)	1
SHMALL	Total amount of shared memory available (bytes or pages)	if bytes, same as SHMMAX; if pages, ceil(SHMMAX/PAGE_SIZE)
SHMSEG	Maximum number of shared memory segments per process	only 1 segment is needed, but the default is much higher
SHMMNI	Maximum number of shared memory segments system-wide	like SHMSEG plus room for other applications
SEMMNI	Maximum number of semaphore identifiers (i.e., sets)	>= ceil(max_connections / 16)
SEMMNS	Maximum number of semaphores system-wide	ceil(max_connections / 16) * 17 + room for other applications
SEMMSL	Maximum number of semaphores per set	>= 17
SEMMAP	Number of entries in semaphore map	see text
SEMVMX	Maximum value of semaphore	>= 255 (The default is often 32767, don't change unless asked to.)

The most important shared memory parameter is SHMMAX, the maximum size, in bytes, that a shared memory segment can have. If you get an error message from shmget along the lines of Invalid argument then it is possible that this limit has been exceeded. The size of the required shared memory segments varies both with the number of requested buffers (-B option) and the number of allowed connections (-N option), although the former is the dominant item. (You can therefore, as a temporary solution, lower these settings to get rid of the failures.) As a rough approximation you can estimate the required segment size as the number of buffers times the block size (8 kB by default) plus ample overhead (at least half a megabyte). Any error message you might get will contain the size of the failed allocation request.

Less likely to cause problems is the minimum size for shared memory segments (SHMMIN), which should be at most somewhere around 256 kB for PostgreSQL (it is usually just 1). The maximum number of segments system-wide (SHMMNI) or per-process (SHMSEG) should not cause a problem unless your system has them set to zero. Some systems also have a limit on the total amount of shared memory in the system; see the platform-specific instructions below.

PostgreSQL uses one semaphore per allowed connection (`-N` option), in sets of 16. Each such set will also contain a 17th semaphore which contains a "magic number", to detect collision with semaphore sets used by other applications. The maximum number of semaphores in the system is set by `SEMMNS`, which consequently must be at least as high as the connection setting plus one extra for each 16 allowed connections (see the formula in Table 3.2, "System V IPC parameters"). The parameter `SEMMNI` determines the limit on the number of semaphore sets that can exist on the system at one time. Hence this parameter must be at least `ceil(max_connections / 16)`. Lowering the number of allowed connections is a temporary workaround for failures, which are usually confusingly worded "No space left on device", from the function `semget()`.

In some cases it might also turn out to be necessary to increase `SEMMAP` to be at least on the order of `SEMMNS`. This parameter defines the size of the semaphore resource map, in which each contiguous block of available semaphores needs an entry. When a semaphore set is freed it is either added to an existing entry that is adjacent to the freed block or it is registered under a new map entry. If the map is full, the freed semaphores get lost (until reboot). Fragmentation of the semaphore space could therefore over time lead to less available semaphores than there should be.

The `SEMMSL` parameter, which determines how many semaphores can be in a set, must be at least 17 for PostgreSQL.

Various other settings related to "semaphore undo", such as `SEMMNU` and `SEMUME`, are not of concern for PostgreSQL.

BSD/OS

Shared Memory. By default, only 4 MB of shared memory is supported. Keep in mind that shared memory is not pageable; it is locked in RAM. To increase the number of shared buffers supported by the postmaster, add the following to your kernel configuration file. A `SHMALL` value of 1024 represents 4MB of shared memory. The following increases the maximum shared memory area to 32 MB:

```
options "SHMALL=8192"
options "SHMMAX=\(SHMALL*PAGE_SIZE\)"
```

For those running 4.1 or later, just make the above changes, recompile the kernel, and reboot. For those running earlier releases, use bpatch to find the `sysptsize` value in the current kernel. This is computed dynamically at boot time.

```
$ bpatch -r sysptsize
0x9 = 9
```

Next, add `SYSPTSIZE` as a hard-coded value in the kernel configuration file. Increase the value you found using bpatch. Add 1 for every additional 4 MB of shared memory you desire.

```
options "SYSPTSIZE=16"
```

`sysptsize` cannot be changed by `sysctl`.

Semaphores. You may need to increase the number of semaphores. By default, PostgreSQL allocates 34 semaphores, which is over half the default system total of 60.

Set the values you want in your kernel configuration file, e.g.:

```
options "SEMMNI=40"
options "SEMMNS=240"
options "SEMUME=40"
```

```
options "SEMMNU=120"
```

FreeBSD
NetBSD
OpenBSD

The options `SYSVSHM` and `SYSVSEM` need to be enabled when the kernel is compiled. (They are by default.) The maximum size of shared memory is determined by the option `SHMMAXPGS` (in pages). The following shows an example of how to set the various parameters:

```
options         SYSVSHM
options         SHMMAXPGS=4096
options         SHMSEG=256

options         SYSVSEM
options         SEMMNI=256
options         SEMMNS=512
options         SEMMNU=256
options         SEMMAP=256
```

(On NetBSD and OpenBSD the key word is actually `option` singular.)

HP-UX

The default settings tend to suffice for normal installations. On HP-UX 10, the factory default for `SEMMNS` is 128, which might be too low for larger database sites.

IPC parameters can be set in the System Administration Manager (SAM) under Kernel Configuration → Configurable Parameters. Hit Create A New Kernel when you're done.

Linux

The default shared memory limit (both `SHMMAX` and `SHMALL`) is 32 MB in 2.2 kernels, but it can be changed in the `proc` file system (without reboot). For example, to allow 128 MB:

```
$ echo 134217728 >/proc/sys/kernel/shmall
$ echo 134217728 >/proc/sys/kernel/shmmax
```

You could put these commands into a script run at boot-time.

Alternatively, you can use sysctl, if available, to control these parameters. Look for a file called `/etc/sysctl.conf` and add lines like the following to it:

```
kernel.shmall = 134217728
kernel.shmmax = 134217728
```

This file is usually processed at boot time, but sysctl can also be called explicitly later.

Other parameters are sufficiently sized for any application. If you want to see for yourself look into `/usr/src/linux/include/asm-`*xxx*`/shmparam.h` and `/usr/src/linux/include/linux/sem.h`.

SCO OpenServer

In the default configuration, only 512 kB of shared memory per segment is allowed, which is about enough for `-B 24 -N 12`. To increase the setting, first change the directory to `/etc/conf/cf.d`. To display the current value of `SHMMAX`, in bytes, run

```
./configure -y SHMMAX
```

To set a new value for `SHMMAX`, run:

```
./configure SHMMAX=value
```

where *`value`* is the new value you want to use (in bytes). After setting `SHMMAX`, rebuild the kernel

```
./link_unix
```

and reboot.

Solaris

At least in version 2.6, the maximum size of a shared memory segment is set too low for PostgreSQL. The relevant settings can be changed in `/etc/system`, for example:

```
set shmsys:shminfo_shmmax=0x2000000
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=256
set shmsys:shminfo_shmseg=256

set semsys:seminfo_semmap=256
set semsys:seminfo_semmni=512
set semsys:seminfo_semmns=512
set semsys:seminfo_semmsl=32
```

You need to reboot to make the changes effective.

See also http://www.sunworld.com/swol-09-1997/swol-09-insidesolaris.html for information on shared memory under Solaris.

UnixWare

On UnixWare 7, the maximum size for shared memory segments is 512 kB in the default configuration. This is enough for about `-B 24 -N 12`. To display the current value of `SHMMAX`, run

```
/etc/conf/bin/idtune -g SHMMAX
```

which displays the current, default, minimum, and maximum values, in bytes. To set a new value for `SHMMAX`, run:

```
/etc/conf/bin/idtune SHMMAX value
```

where *`value`* is the new value you want to use (in bytes). After setting `SHMMAX`, rebuild the kernel

```
/etc/conf/bin/idbuild -B
```

and reboot.

3.5.2. Resource Limits

Unix-like operating systems enforce various kinds of resource limits that might interfere with the operation of your PostgreSQL server. Of importance are especially the limits on the number of processes per user, the number of open files per process, and the amount of memory available to a process. Each of these have a “hard” and a “soft” limit. The soft limit is what actually counts but it can be changed by the user up to the hard limit. The hard limit can only be changed by the root user. The system call `setrlimit` is

responsible for setting these parameters. The shell's built-in command `ulimit` (Bourne shells) or `limit` (csh) is used to control the resource limits from the command line. On BSD-derived systems the file `/etc/login.conf` controls what values the various resource limits are set to upon login. See login.conf for details. The relevant parameters are `maxproc`, `openfiles`, and `datasize`. For example:

```
default:\
...
        :datasize-cur=256M:\
        :maxproc-cur=256:\
        :openfiles-cur=256:\
...
```

(`-cur` is the soft limit. Append `-max` to set the hard limit.)

Kernels generally also have an implementation-dependent system-wide limit on some resources.

- On Linux `/proc/sys/fs/file-max` determines the maximum number of open files that the kernel will support. It can be changed by writing a different number into the file or by adding an assignment in `/etc/sysctl.conf`. The maximum limit of files per process is fixed at the time the kernel is compiled; see `/usr/src/linux/Documentation/proc.txt` for more information.

The PostgreSQL server uses one process per connection so you should provide for at least as many processes as allowed connections, in addition to what you need for the rest of your system. This is usually not a problem but if you run several servers on one machine things might get tight.

The factory default limit on open files is often set to “socially friendly” values that allow many users to coexist on a machine without using an inappropriate fraction of the system resources. If you run many servers on a machine this is perhaps what you want, but on dedicated servers you may want to raise this limit.

On the other side of the coin, some systems allow individual processes to open large numbers of files; if more than a few processes do so then the system-wide limit can easily be exceeded. If you find this happening, and don't want to alter the system-wide limit, you can set PostgreSQL's `max_files_per_process` configuration parameter to limit its consumption of open files.

3.6. Shutting down the server

Depending on your needs, there are several ways to shut down the database server when your work is done. The differentiation is done by what signal you send to the server process.

SIGTERM

After receiving SIGTERM, the postmaster disallows new connections, but lets existing backends end their work normally. It shuts down only after all of the backends terminate by client request. This is the *Smart Shutdown*.

SIGINT

The postmaster disallows new connections and sends all existing backends SIGTERM, which will cause them to abort their current transactions and exit promptly. It then waits for the backends to exit and finally shuts down the data base. This is the *Fast Shutdown*.

SIGQUIT

This is the *Immediate Shutdown* which will cause the postmaster to send a SIGQUIT to all backends and exit immediately (without properly shutting down the database system). The backends likewise

exit immediately upon receiving SIGQUIT. This will lead to recovery (by replaying the WAL log) upon next start-up. This is recommended only in emergencies.

Important

It is best not to use SIGKILL to shut down the postmaster. This will prevent the postmaster from releasing shared memory and semaphores, which you may then have to do by hand.

The PID of the postmaster process can be found using the ps program, or from the file `postmaster.pid` in the data directory. So for example, to do a fast shutdown:

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

The program pg_ctl is a shell script that provides a more convenient interface for shutting down the postmaster.

3.7. Secure TCP/IP Connections with SSL

PostgreSQL has native support for connections over SSL to encrypt client/server communications for increased security. This requires OpenSSL to be installed on both client and server systems and support enabled at build time (see Chapter 1, *Installation Instructions*).

With SSL support compiled in, the PostgreSQL server can be started with the argument `-l` (ell) to enable SSL connections. When starting in SSL mode, the server will look for the files `server.key` and `server.crt` in the data directory. These files should contain the server private key and certificate respectively. These files must be set up correctly before an SSL-enabled server can start. If the private key is protected with a passphrase, the server will prompt for the passphrase and will not start until it has been entered.

The server will listen for both standard and SSL connections on the same TCP/IP port, and will negotiate with any connecting client whether or not to use SSL. See Chapter 4, *Client Authentication* about how to force on the server side the use of SSL for certain connections.

For details on how to create your server private key and certificate, refer to the OpenSSL documentation. A simple self-signed certificate can be used to get started for testing, but a certificate signed by a CA (either one of the global CAs or a local one) should be used in production so the client can verify the server's identity. To create a quick self-signed certificate, use the following OpenSSL command:

```
openssl req -new -text -out cert.req
```

Fill out the information that `openssl` asks for. Make sure that you enter the local host name as Common Name; the challenge password can be left blank. The script will generate a key that is passphrase protected; it will not accept a pass phrase that is less than four characters long. To remove the passphrase (as you must if you want automatic start-up of the server), run the commands

```
openssl rsa -in privkey.pem -out cert.pem
```

Enter the old passphrase to unlock the existing key. Now do

```
openssl req -x509 -in cert.req -text -key cert.pem -out cert.cert
cp cert.pem $PGDATA/server.key
cp cert.cert $PGDATA/server.crt
```

to turn the certificate into a self-signed certificate and to copy the key and certificate to where the server will look for them.

3.8. Secure TCP/IP Connections with SSH tunnels

Acknowledgement

Idea taken from an email by Gene Selkov, Jr. (<selkovjr@mcs.anl.gov>) written on 1999-09-08 in response to a question from Eric Marsden.

One can use ssh to encrypt the network connection between clients and a PostgreSQL server. Done properly, this should lead to an adequately secure network connection.

First make sure that an ssh server is running properly on the same machine as PostgreSQL and that you can log in using `ssh` as some user. Then you can establish a secure tunnel with a command like this from the client machine:

```
$ ssh -L 3333:foo.com:5432 joe@foo.com
```

The first number in the `-L` argument, 3333, is the port number of your end of the tunnel; it can be chosen freely. The second number, 5432, is the remote end of the tunnel -- the port number your server is using. The name or the address in between the port numbers is the host with the database server you are going to connect to. In order to connect to the database server using this tunnel, you connect to port 3333 on the local machine:

```
psql -h localhost -p 3333 template1
```

To the database server it will then look as though you are really user `joe@foo.com` and it will use whatever authentication procedure was set up for this user. In order for the tunnel setup to succeed you must be allowed to connect via `ssh` as joe@foo.com, just as if you had attempted to use `ssh` to set up a terminal session.

Tip

Several other products exist that can provide secure tunnels using a procedure similar in concept to the one just described.

Chapter 4. Client Authentication

When a client application connects to the database server, it specifies which PostgreSQL user name it wants to connect as, much the same way one logs into a Unix computer as a particular user. Within the SQL environment the active database user name determines access privileges to database objects -- see Chapter 7, *Database Users and Permissions* for more information about that. It is therefore obviously essential to restrict which database user name(s) a given client can connect as.

Authentication is the process by which the database server establishes the identity of the client, and by extension determines whether the client application (or the user who runs the client application) is permitted to connect with the user name that was requested.

PostgreSQL offers a number of different client authentication methods. The method to be used can be selected on the basis of (client) host and database; some authentication methods allow you to restrict by user name as well.

PostgreSQL database user names are logically separate from user names of the operating system in which the server runs. If all the users of a particular server also have accounts on the server's machine, it makes sense to assign database user names that match their operating system user names. However, a server that accepts remote connections may have many users who have no local account, and in such cases there need be no connection between database user names and OS user names.

4.1. The `pg_hba.conf` file

Client authentication is controlled by the file `pg_hba.conf` in the data directory, e.g., `/usr/local/pgsql/data/pg_hba.conf`. (HBA stands for host-based authentication.) A default `pg_hba.conf` file is installed when the data area is initialized by `initdb`.

The general format of the `pg_hba.conf` file is of a set of records, one per line. Blank lines and lines beginning with a hash character ("#") are ignored. A record is made up of a number of fields which are separated by spaces and/or tabs. Records cannot be continued across lines.

Each record specifies a connection type, a client IP address range (if relevant for the connection type), a database name or names, and the authentication method to be used for connections matching these parameters. The first record that matches the type, client address, and requested database name of a connection attempt is used to do the authentication step. There is no "fall-through" or "backup": if one record is chosen and the authentication fails, the following records are not considered. If no record matches, the access will be denied.

A record may have one of the three formats

```
local   database authentication-method [ authentication-option ]
host    database IP-address IP-mask authentication-method
 [ authentication-option ]
hostssl database IP-address IP-mask authentication-method
 [ authentication-option ]
```

The meaning of the fields is as follows:

`local`

This record pertains to connection attempts over Unix domain sockets.

`host`

This record pertains to connection attempts over TCP/IP networks. Note that TCP/IP connections are completely disabled unless the server is started with the `-i` switch or the equivalent configuration parameter is set.

`hostssl`

This record pertains to connection attempts with SSL over TCP/IP. To make use of this option the server must be built with SSL support enabled. Furthermore, SSL must be enabled with the `-l` option or equivalent configuration setting when the server is started. (Note: `host` records will match either SSL or non-SSL connection attempts, but `hostssl` records match only SSL connections.)

`database`

Specifies the database that this record applies to. The value `all` specifies that it applies to all databases, while the value `sameuser` identifies the database with the same name as the connecting user. Otherwise, this is the name of a specific PostgreSQL database.

`IP address`
`IP mask`

These two fields specify to which client machines a `host` or `hostssl` record applies, based on their IP address. (Of course IP addresses can be spoofed but this consideration is beyond the scope of PostgreSQL.) The precise logic is that

```
(actual-IP-address xor IP-address-field) and IP-mask-
field
```

must be zero for the record to match.

`authentication method`

Specifies the method that users must use to authenticate themselves when connecting under the control of this authentication record. The possible choices are summarized here, details are in Section 4.2, “Authentication methods”.

`trust`

The connection is allowed unconditionally. This method allows any user that has login access to the client host to connect as any PostgreSQL user whatsoever.

`reject`

The connection is rejected unconditionally. This is mostly useful to “filter out” certain hosts from a group.

`password`

The client is required to supply a password which is required to match the database password that was set up for the user.

An optional file name may be specified after the `password` keyword. This file is expected to contain a list of users who may connect using this record, and optionally alternative passwords for them.

The password is sent over the wire in clear text. For better protection, use the `md5` or `crypt` methods.

`md5`

Like the `password` method, but the password is sent over the wire encrypted using a simple challenge-response protocol. This protects against incidental wire-sniffing. This is now the recommended choice for password-based authentication.

The name of a file may follow the `md5` keyword. It contains a list of users who may connect using this record.

`crypt`

Like the `md5` method but uses older crypt encryption, which is needed for pre-7.2 clients. `md5` is preferred for 7.2 and later clients. The `crypt` method is not compatible with encrypting passwords in `pg_shadow`, and may fail if client and server machines have different implementations of the crypt() library routine.

`krb4`

Kerberos V4 is used to authenticate the user. This is only available for TCP/IP connections.

`krb5`

Kerberos V5 is used to authenticate the user. This is only available for TCP/IP connections.

`ident`

The identity of the user as determined on login to the operating system is used by PostgreSQL to determine whether the user is allowed to connect as the requested database user. For TCP/IP connections the user's identity is determined by contacting the *ident* server on the client host. (Note that this is only as reliable as the remote ident server; ident authentication should never be used for remote hosts whose administrators are not trustworthy.) On operating systems supporting `SO_PEERCRED` requests for Unix domain sockets, ident authentication is possible for local connections; the system is then asked for the connecting user's identity.

On systems without `SO_PEERCRED` requests, ident authentication is only available for TCP/IP connections. As a workaround, it is possible to specify the localhost address 127.0.0.1 and make connections to this address.

The *`authentication option`* following the `ident` keyword specifies the name of an *ident map* that specifies which operating system users equate with which database users. See below for details.

`pam`

This authentication type operates similarly to *password*, with the main difference that it will use PAM (Pluggable Authentication Modules) as the authentication mechanism. The *`authentication option`* following the `pam` keyword specifies the service name that will be passed to PAM. The default service name is `postgresql`. For more information about PAM, please read the Linux-PAM Page[1] and/or the Solaris PAM Page[2].

`authentication option`

This field is interpreted differently depending on the authentication method, as described above.

[1] http://www.kernel.org/pub/linux/libs/pam/
[2] http://www.sun.com/software/solaris/pam/

Since the pg_hba.conf records are examined sequentially for each connection attempt, the order of the records is very significant. Typically, earlier records will have tight connection match parameters and weaker authentication methods, while later records will have looser match parameters and stronger authentication methods. For example, one might wish to use trust authentication for local TCP connections but require a password for remote TCP connections. In this case a record specifying trust authentication for connections from 127.0.0.1 would appear before a record specifying password authentication for a wider range of allowed client IP addresses.

The pg_hba.conf file is read on startup and when the postmaster receives a SIGHUP signal. If you edit the file on an active system, you will need to signal the postmaster (using pg_ctl reload or kill -HUP) to make it re-read the file.

An example of a pg_hba.conf file is shown in Example 4.1, "An example pg_hba.conf file". See below for details on the different authentication methods.

Example 4.1. An example pg_hba.conf file

```
# TYPE       DATABASE    IP_ADDRESS    MASK               AUTHTYPE
 MAP

# Allow any user on the local system to connect to any
# database under any username, but only via an IP connection:

host         all         127.0.0.1     255.255.255.255    trust

# The same, over Unix-socket connections:

local        all                                          trust

# Allow any user from any host with IP address 192.168.93.x to
# connect to database "template1" as the same username that ident on
 that
# host identifies him as (typically his Unix username):

host         template1   192.168.93.0  255.255.255.0      ident
 sameuser

# Allow a user from host 192.168.12.10 to connect to database
 "template1"
# if the user's password in pg_shadow is correctly supplied:

host         template1   192.168.12.10 255.255.255.255    md5

# In the absence of preceding "host" lines, these two lines will
 reject
# all connection attempts from 192.168.54.1 (since that entry will be
# matched first), but allow Kerberos V5-validated connections from
 anywhere
# else on the Internet. The zero mask means that no bits of the host
 IP
# address are considered, so it matches any host:

host         all        192.168.54.1   255.255.255.255    reject
host         all        0.0.0.0        0.0.0.0            krb5
```

```
# Allow users from 192.168.x.x hosts to connect to any database, if
 they
# pass the ident check.  If, for example, ident says the user is
 "bryanh"
# and he requests to connect as PostgreSQL user "guest1", the
 connection
# is allowed if there is an entry in pg_ident.conf for map "omicron"
 that
# says "bryanh" is allowed to connect as "guest1":

host         all        192.168.0.0    255.255.0.0        ident
 omicron

# If these are the only two lines for local connections, they will
 allow
# local users to connect only to their own databases (database named
 the
# same as the user name), except for administrators who may connect to
# all databases.  The file $PGDATA/admins lists the user names who are
# permitted to connect to all databases.  Passwords are required in
 all
# cases.  (If you prefer to use ident authorization, an ident map can
# serve a parallel purpose to the password list file used here.)

local        sameuser                                    md5
local        all                                         md5  admins
```

4.2. Authentication methods

The following describes the authentication methods in more detail.

4.2.1. Trust authentication

When `trust` authentication is specified, PostgreSQL assumes that anyone who can connect to the postmaster is authorized to access the database as whatever database user he specifies (including the database superuser). This method should only be used when there is adequate system-level protection on connections to the postmaster port.

`trust` authentication is appropriate and very convenient for local connections on a single-user workstation. It is usually *not* appropriate by itself on a multiuser machine. However, you may be able to use `trust` even on a multiuser machine, if you restrict access to the postmaster's socket file using file-system permissions. To do this, set the parameter `unix_socket_permissions` (and possibly `unix_socket_group`) in `postgresql.conf`, as described in Section 3.4.3, “General operation”. Or you could set `unix_socket_directory` to place the socket file in a suitably restricted directory.

Setting file-system permissions only helps for Unix-socket connections. Local TCP connections are not restricted by it; therefore, if you want to use permissions for local security, remove the `host ... 127.0.0.1 ...` line from `pg_hba.conf`, or change it to a non-`trust` authentication method.

`trust` authentication is only suitable for TCP connections if you trust every user on every machine that is allowed to connect to the postmaster by the `pg_hba.conf` lines that specify `trust`. It is seldom reasonable to use `trust` for any TCP connections other than those from localhost (127.0.0.1).

4.2.2. Password authentication

Password-based authentication methods include `md5`, `crypt`, and `password`. These methods operate similarly except for the way that the password is sent across the connection. If you are at all concerned about password "sniffing" attacks then `md5` is preferred, with `crypt` a second choice if you must support obsolete clients. Plain `password` should especially be avoided for connections over the open Internet (unless you use SSL, SSH, or other communications security wrappers around the connection).

PostgreSQL database passwords are separate from operating system user passwords. Ordinarily, the password for each database user is stored in the pg_shadow system catalog table. Passwords can be managed with the query language commands `CREATE USER` and `ALTER USER`, e.g., **`CREATE USER foo WITH PASSWORD 'secret';`**. By default, that is, if no password has been set up, the stored password is `NULL` and password authentication will always fail for that user.

To restrict the set of users that are allowed to connect to certain databases, list the set of users in a separate file (one user name per line) in the same directory that `pg_hba.conf` is in, and mention the (base) name of the file after the `password`, `md5`, or `crypt` keyword, respectively, in `pg_hba.conf`. If you do not use this feature, then any user that is known to the database system can connect to any database (so long as he supplies the correct password, of course).

These files can also be used to apply a different set of passwords to a particular database or set thereof. In that case, the files have a format similar to the standard Unix password file `/etc/passwd`, that is,

username:*password*

Any extra colon-separated fields following the password are ignored. The password is expected to be encrypted using the system's `crypt()` function. The utility program pg_passwd that is installed with PostgreSQL can be used to manage these password files.

Lines with and without passwords can be mixed in secondary password files. Lines without password indicate use of the main password in `pg_shadow` that is managed by `CREATE USER` and `ALTER USER`. Lines with passwords will cause that password to be used. A password entry of "+" also means using the pg_shadow password.

Alternative passwords cannot be used when using the `md5` or `crypt` methods. The file will be read as usual, but the password field will simply be ignored and the `pg_shadow` password will always be used.

Note that using alternative passwords like this means that one can no longer use `ALTER USER` to change one's password. It will appear to work but the password one is changing is not the password that the system will end up using.

4.2.3. Kerberos authentication

Kerberos is an industry-standard secure authentication system suitable for distributed computing over a public network. A description of the Kerberos system is far beyond the scope of this document; in all generality it can be quite complex (yet powerful). The Kerberos FAQ[3] or MIT Project Athena[4] can be a good starting point for exploration. Several sources for Kerberos distributions exist.

In order to use Kerberos, support for it must be enabled at build time. Both Kerberos 4 and 5 are supported (`./configure --with-krb4` or `./configure --with-krb5` respectively), although only one version can be supported in any one build.

[3] http://www.nrl.navy.mil/CCS/people/kenh/kerberos-faq.html
[4] ftp://athena-dist.mit.edu

PostgreSQL operates like a normal Kerberos service. The name of the service principal is *servicename/hostname@realm*, where *servicename* is `postgres` (unless a different service name was selected at configure time with `./configure --with-krb-srvnam=whatever`). *hostname* is the fully qualified domain name of the server machine. The service principal's realm is the preferred realm of the server machine.

Client principals must have their PostgreSQL user name as their first component, for example *pgusername/otherstuff@realm*. At present the realm of the client is not checked by PostgreSQL; so if you have cross-realm authentication enabled, then any principal in any realm that can communicate with yours will be accepted.

Make sure that your server key file is readable (and preferably only readable) by the PostgreSQL server account (see Section 3.1, "The PostgreSQL user account"). The location of the key file is specified with the `krb_server_keyfile` run time configuration parameter. (See also Section 3.4, "Run-time configuration".) The default is `/etc/srvtab` if you are using Kerberos 4 and `FILE:/usr/local/pgsql/etc/krb5.keytab` (or whichever directory was specified as `sysconfdir` at build time) with Kerberos 5.

To generate the keytab file, use for example (with version 5)

```
kadmin% ank -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

Read the Kerberos documentation for details.

When connecting to the database make sure you have a ticket for a principal matching the requested database user name. An example: For database user name `fred`, both principal `fred@EXAMPLE.COM` and `fred/users.example.com@EXAMPLE.COM` can be used to authenticate to the database server.

If you use mod_auth_krb and mod_perl on your Apache web server, you can use `AuthType KerberosV5SaveCredentials` with a mod_perl script. This gives secure database access over the web, no extra passwords required.

4.2.4. Ident-based authentication

The "Identification Protocol" is described in *RFC 1413*. Virtually every Unix-like operating system ships with an ident server that listens on TCP port 113 by default. The basic functionality of an ident server is to answer questions like "What user initiated the connection that goes out of your port *X* and connects to my port *Y*?". Since PostgreSQL knows both *X* and *Y* when a physical connection is established, it can interrogate the ident server on the host of the connecting client and could theoretically determine the operating system user for any given connection this way.

The drawback of this procedure is that it depends on the integrity of the client: if the client machine is untrusted or compromised an attacker could run just about any program on port 113 and return any user name he chooses. This authentication method is therefore only appropriate for closed networks where each client machine is under tight control and where the database and system administrators operate in close contact. In other words, you must trust the machine running the ident server. Heed the warning:

> The Identification Protocol is not intended as an authorization or access control protocol.
>
> —RFC 1413

On systems supporting `SO_PEERCRED` requests for Unix-domain sockets, ident authentication can also be applied to local connections. In this case, no security risk is added by using ident authentication; indeed it is a preferable choice for local connections on such a system.

When using ident-based authentication, after having determined the name of the operating system user that initiated the connection, PostgreSQL checks whether that user is allowed to connect as the database user he

is requesting to connect as. This is controlled by the ident map argument that follows the `ident` keyword in the `pg_hba.conf` file. There is a predefined ident map `sameuser`, which allows any operating system user to connect as the database user of the same name (if the latter exists). Other maps must be created manually.

Ident maps other than `sameuser` are defined in the file `pg_ident.conf` in the data directory, which contains lines of the general form:

```
map-name ident-username database-username
```

Comments and whitespace are handled in the usual way. The *`map-name`* is an arbitrary name that will be used to refer to this mapping in `pg_hba.conf`. The other two fields specify which operating system user is allowed to connect as which database user. The same *`map-name`* can be used repeatedly to specify more user-mappings within a single map. There is no restriction regarding how many database users a given operating system user may correspond to and vice versa.

The `pg_ident.conf` file is read on startup and when the postmaster receives a SIGHUP signal. If you edit the file on an active system, you will need to signal the postmaster (using `pg_ctl reload` or `kill -HUP`) to make it re-read the file.

A `pg_ident.conf` file that could be used in conjunction with the `pg_hba.conf` file in Example 4.1, "An example `pg_hba.conf` file" is shown in Example 4.2, "An example `pg_ident.conf` file". In this example setup, anyone logged in to a machine on the 192.168 network that does not have the Unix user name bryanh, ann, or robert would not be granted access. Unix user robert would only be allowed access when he tries to connect as PostgreSQL user bob, not as robert or anyone else. ann would only be allowed to connect as ann. User bryanh would be allowed to connect as either bryanh himself or as guest1.

Example 4.2. An example `pg_ident.conf` file

```
#MAP            IDENT-NAME    POSTGRESQL-NAME

omicron         bryanh        bryanh
omicron         ann           ann
# bob has username robert on these machines
omicron         robert        bob
# bryanh can also connect as guest1
omicron         bryanh        guest1
```

4.3. Authentication problems

Genuine authentication failures and related problems generally manifest themselves through error messages like the following.

```
No pg_hba.conf entry for host 123.123.123.123, user joeblow, database
 testdb
```

This is what you are most likely to get if you succeed in contacting the server, but it does not want to talk to you. As the message suggests, the server refused the connection request because it found no authorizing entry in its `pg_hba.conf` configuration file.

```
Password authentication failed for user 'joeblow'
```

Messages like this indicate that you contacted the server, and it is willing to talk to you, but not until you pass the authorization method specified in the `pg_hba.conf` file. Check the password you are providing, or check your Kerberos or ident software if the complaint mentions one of those authentication types.

```
FATAL 1:  user "joeblow" does not exist
```

The indicated user name was not found.

```
FATAL 1:  Database "testdb" does not exist in the system catalog.
```

The database you are trying to connect to does not exist. Note that if you do not specify a database name, it defaults to the database user name, which may or may not be the right thing.

Note that the server log may contain more information about an authentication failure than is reported to the client. If you are confused about the reason for a failure, check the log.

Chapter 5. Localization

Describes the available localization features from the point of view of the administrator.

PostgreSQL supports localization with three approaches:

- Using the locale features of the operating system to provide locale-specific collation order, number formatting, translated messages, and other aspects.
- Using explicit multiple-byte character sets defined in the PostgreSQL server to support languages that require more characters than will fit into a single byte, and to provide character set recoding between client and server. The number of supported character sets is fixed at the time the server is compiled, and internal operations such as string comparisons require expansion of each character into a 32-bit word.
- Single byte character recoding provides a more light-weight solution for users of multiple, yet single-byte character sets.

5.1. Locale Support

Locale support refers to an application respecting cultural preferences regarding alphabets, sorting, number formatting, etc. PostgreSQL uses the standard ISO C and POSIX-like locale facilities provided by the server operating system. For additional information refer to the documentation of your system.

5.1.1. Overview

Locale support is not built into PostgreSQL by default; to enable it, supply the `--enable-locale` option to the `configure` script:

```
$ ./configure --enable-locale
```

Locale support only affects the server; all clients are compatible with servers with or without locale support.

To enable messages translated to the user's preferred language, the `--enable-nls` option must be used. This option is independent of the other locale support.

The information about which particular cultural rules to use is determined by standard environment variables. If you are getting localized behavior from other programs you probably have them set up already. The simplest way to set the localization information is the `LANG` variable, for example:

```
export LANG=sv_SE
```

This sets the locale to Swedish (`sv`) as spoken in Sweden (`SE`). Other possibilities might be `en_US` (U.S. English) and `fr_CA` (Canada, French). If more than one character set can be useful for a locale then the specifications look like this: `cs_CZ.ISO8859-2`. What locales are available under what names on your system depends on what was provided by the operating system vendor and what was installed.

Occasionally it is useful to mix rules from several locales, e.g., use U.S. collation rules but Spanish messages. To do that a set of environment variables exist that override the default of `LANG` for a particular category:

`LC_COLLATE`	String sort order

LC_CTYPE	Character classification (What is a letter? The upper-case equivalent?)
LC_MESSAGES	Language of messages
LC_MONETARY	Formatting of currency amounts
LC_NUMERIC	Formatting of numbers
LC_TIME	Formatting of dates and times

Additionally, all of these specific variables and the LANG variable can be overridden with the LC_ALL environment variable.

Note

Some message localization libraries also look at the environment variable LANGUAGE which overrides all other locale settings for the purpose of setting the language of messages. If in doubt, please refer to the documentation of your operating system, in particular the gettext manual page, for more information.

If you want the system to behave as if it had no locale support, use the special locale C or POSIX, or simply unset all locale-related variables.

Note that the locale behavior of the server is determined by the environment variables seen by the server, not by the environment of any client. Therefore, be careful to set these variables before starting the server. A consequence of this is that if client and server are set up to different locales, messages may appear in different languages depending on where they originated.

The LC_COLLATE and LC_CTYPE variables affect the sort order of indexes. Therefore, these values must be kept fixed for any particular database cluster, or indexes on text columns will become corrupt. PostgreSQL enforces this by recording the values of LC_COLLATE and LC_CTYPE that are seen by initdb. The server automatically adopts those two values when it is started; only the other LC_ categories can be set from the environment at server startup. In short, only one collation order can be used in a database cluster, and it is chosen at initdb time.

5.1.2. Benefits

Locale support influences in particular the following features:

- Sort order in ORDER BY queries.
- The to_char family of functions
- The LIKE and ~ operators for pattern matching

The only severe drawback of using the locale support in PostgreSQL is its speed. So use locale only if you actually need it. It should be noted in particular that selecting a non-C locale disables index optimizations for LIKE and ~ operators, which can make a huge difference in the speed of searches that use those operators.

5.1.3. Problems

If locale support doesn't work in spite of the explanation above, check that the locale support in your operating system is correctly configured. To check whether a given locale is installed and functional you

can use Perl, for example. Perl has also support for locales and if a locale is broken `perl -v` will complain something like this:

```
$ export LC_CTYPE='not_exist'
$ perl -v
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
LC_ALL = (unset),
LC_CTYPE = "not_exist",
LANG = (unset)
are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

Check that your locale files are in the right location. Possible locations include: `/usr/lib/locale` (Linux, Solaris), `/usr/share/locale` (Linux), `/usr/lib/nls/loc` (DUX 4.0). Check the locale man page of your system if you are not sure.

Check that PostgreSQL is actually using the locale that you think it is. `LC_COLLATE` and `LC_CTYPE` settings are determined at initdb time and cannot be changed without repeating initdb. Other locale settings including `LC_MESSAGES` and `LC_MONETARY` are determined by the environment the postmaster is started in, and can be changed with a simple postmaster restart. You can check the `LC_COLLATE` and `LC_CTYPE` settings of a database with the `contrib/pg_controldata` utility program.

The directory `src/test/locale` contains a test suite for PostgreSQL's locale support.

Client applications that handle server-side errors by parsing the text of the error message will obviously have problems when the server's messages are in a different language. If you create such an application you need to devise a plan to cope with this situation. The embedded SQL interface (ecpg) is also affected by this problem. It is currently recommended that servers interfacing with ecpg applications be configured to send messages in English.

Maintaining catalogs of message translations requires the on-going efforts of many volunteers that want to see PostgreSQL speak their preferred language well. If messages in your language is currently not available or fully translated, your assistance would be appreciated. If you want to help, refer to the *Developer's Guide* or write to the developers' mailing list.

5.2. Multibyte Support

Author

Tatsuo Ishii (<ishii@postgresql.org>), last updated 2000-03-22. Check Tatsuo's web site[1] for more information.

Multibyte (MB) support is intended to allow PostgreSQL to handle multiple-byte character sets such as EUC (Extended Unix Code), Unicode, and Mule internal code. With MB enabled you can use multibyte character sets in regular expressions (regexp), LIKE, and some other functions. The default encoding system is selected while initializing your PostgreSQL installation using initdb. Note that this can be overridden when you create a database using createdb or by using the SQL command `CREATE DATABASE`. So you can have multiple databases each with a different encoding system.

5.2.1. Enabling Multibyte Support

Run configure with the multibyte option:

[1] http://www.sra.co.jp/people/t-ishii/PostgreSQL/

```
./configure --enable-multibyte[=encoding_system]
```

where *encoding_system* can be one of the values in the following table:

Table 5.1. Character Set Encodings

Encoding	Description
SQL_ASCII	ASCII
EUC_JP	Japanese EUC
EUC_CN	Chinese EUC
EUC_KR	Korean EUC
EUC_TW	Taiwan EUC
UNICODE	Unicode (UTF-8)
MULE_INTERNAL	Mule internal code
LATIN1	ISO 8859-1 ECMA-94 Latin Alphabet No.1
LATIN2	ISO 8859-2 ECMA-94 Latin Alphabet No.2
LATIN3	ISO 8859-3 ECMA-94 Latin Alphabet No.3
LATIN4	ISO 8859-4 ECMA-94 Latin Alphabet No.4
LATIN5	ISO 8859-9 ECMA-128 Latin Alphabet No.5
LATIN6	ISO 8859-10 ECMA-144 Latin Alphabet No.6
LATIN7	ISO 8859-13 Latin Alphabet No.7
LATIN8	ISO 8859-14 Latin Alphabet No.8
LATIN9	ISO 8859-15 Latin Alphabet No.9
LATIN10	ISO 8859-16 ASRO SR 14111 Latin Alphabet No.10
ISO-8859-5	ECMA-113 Latin/Cyrillic
ISO-8859-6	ECMA-114 Latin/Arabic
ISO-8859-7	ECMA-118 Latin/Greek
ISO-8859-8	ECMA-121 Latin/Hebrew
KOI8	KOI8-R(U)
WIN	Windows CP1251
ALT	Windows CP866

Important

Before PostgreSQL7.2, LATIN5 mistakenly meant ISO 8859-5. From 7.2 on, LATIN5 means ISO 8859-9. If you have a LATIN5 database created on 7.1 or earlier and want to migrate to 7.2 (or later), you should be very careful about this change.

Important

Not all APIs supports all the encodings listed above. For example, the PostgreSQL JDBC driver does not support MULE_INTERNAL, LATIN6, LATIN8, and LATIN10.

Here is an example of configuring PostgreSQL to use a Japanese encoding by default:

```
$ ./configure --enable-multibyte=EUC_JP
```

If the encoding system is omitted (`./configure --enable-multibyte`), `SQL_ASCII` is assumed.

5.2.2. Setting the Encoding

initdb defines the default encoding for a PostgreSQL installation. For example:

```
$ initdb -E EUC_JP
```

sets the default encoding to `EUC_JP` (Extended Unix Code for Japanese). Note that you can use `--encoding` instead of `-E` if you prefer to type longer option strings. If no `-E` or `--encoding` option is given, the encoding specified at configure time is used.

You can create a database with a different encoding:

```
$ createdb -E EUC_KR korean
```

will create a database named korean with `EUC_KR` encoding. Another way to accomplish this is to use a SQL command:

```
CREATE DATABASE korean WITH ENCODING = 'EUC_KR';
```

The encoding for a database is represented as an *encoding column* in the `pg_database` system catalog. You can see that by using the `-l` option or the `\l` command of `psql`.

```
$ psql -l
            List of databases
   Database    |  Owner   |   Encoding
---------------+---------+---------------
 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
 template1     | t-ishii | EUC_JP
 test          | t-ishii | EUC_JP
 unicode       | t-ishii | UNICODE
(9 rows)
```

5.2.3. Automatic encoding translation between server and client

PostgreSQL supports an automatic encoding translation between server and client for some encodings. The available combinations are listed in Table 5.2, “Communication Encodings”.

Table 5.2. Client/Server Character Set Encodings

Server Encoding	Available Client Encodings
`SQL_ASCII`	`SQL_ASCII`, `UNICODE`, `MULE_INTERNAL`
`EUC_JP`	`EUC_JP`, `SJIS`, `UNICODE`, `MULE_INTERNAL`

Server Encoding	Available Client Encodings
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 MULE_INTERNAL
LATIN6	LATIN6, UNICODE MULE_INTERNAL
LATIN7	LATIN7, UNICODE MULE_INTERNAL
LATIN8	LATIN8, UNICODE MULE_INTERNAL
LATIN9	LATIN9, UNICODE MULE_INTERNAL
LATIN10	LATIN10, UNICODE MULE_INTERNAL
ISO_8859_5	ISO_8859_5, UNICODE
ISO_8859_6	ISO_8859_6, UNICODE
ISO_8859_7	ISO_8859_7, UNICODE
ISO_8859_8	ISO_8859_8, UNICODE
ISO_8859_9	ISO_8859_9, WIN, ALT, KOI8R, UNICODE, MULE_INTERNAL
UNICODE	EUC_JP, SJIS, EUC_KR, EUC_CN, EUC_TW, BIG5, LATIN1 to LATIN10, ISO_8859_5, ISO_8859_6, ISO_8859_7, ISO_8859_8, WIN, ALT, KOI8
MULE_INTERNAL	EUC_JP, SJIS, EUC_KR, EUC_CN, EUC_TW, BIG5, LATIN1 to LATIN5, WIN, ALT, WIN1250
KOI8	ISO_8859_9, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL
WIN	ISO_8859_9, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL
ALT	ISO_8859_9, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL

To enable the automatic encoding translation, you have to tell PostgreSQL the encoding you would like to use in the client. There are several ways to accomplish this.

- Using the `\encoding` command in psql. `\encoding` allows you to change client encoding on the fly. For example, to change the encoding to `SJIS`, type:

  ```
  \encoding SJIS
  ```

- Using libpq functions. `\encoding` actually calls `PQsetClientEncoding()` for its purpose.

  ```
  int PQsetClientEncoding(PGconn *conn, const char *encoding)
  ```

  where *conn* is a connection to the server, and *encoding* is an encoding you want to use. If it successfully sets the encoding, it returns 0, otherwise -1. The current encoding for this connection can be shown by using:

```
int PQclientEncoding(const PGconn *conn)
```

Note that it returns the encoding ID, not a symbolic string such as EUC_JP. To convert an encoding ID to an encoding name, you can use:

```
char *pg_encoding_to_char(int encoding_id)
```

- Using SET CLIENT_ENCODING TO. Setting the client encoding can be done with this SQL command:

  ```
  SET CLIENT_ENCODING TO 'encoding';
  ```

  Also you can use the SQL92 syntax SET NAMES for this purpose:

  ```
  SET NAMES 'encoding';
  ```

  To query the current client encoding:

  ```
  SHOW CLIENT_ENCODING;
  ```

  To return to the default encoding:

  ```
  RESET CLIENT_ENCODING;
  ```

- Using PGCLIENTENCODING. If environment variable PGCLIENTENCODING is defined in the client's environment, that client encoding is automatically selected when a connection to the server is made. (This can subsequently be overridden using any of the other methods mentioned above.)

5.2.4. About Unicode

An automatic encoding translation between Unicode and other encodings has been supported since PostgreSQL 7.1. For 7.1 it was not enabled by default. To enable this feature, run configure with the --enable-unicode-conversion option. Note that this requires the --enable-multibyte option also.

For 7.2, --enable-unicode-conversion is not necessary. The Unicode conversion functionality is automatically enabled if --enable-multibyte is specified.

5.2.5. What happens if the translation is not possible?

Suppose you choose EUC_JP for the server and LATIN1 for the client, then some Japanese characters cannot be translated into LATIN1. In this case, a letter that cannot be represented in the LATIN1 character set would be transformed as:

```
(HEXA DECIMAL)
```

5.2.6. References

These are good sources to start learning about various kinds of encoding systems.

ftp://ftp.ora.com/pub/examples/nutshell/ujip/doc/cjk.inf

Detailed explanations of EUC_JP, EUC_CN, EUC_KR, EUC_TW appear in section 3.2.

http://www.unicode.org/

The web site of the Unicode Consortium

RFC 2044

UTF-8 is defined here.

5.2.7. History

```
Dec 7, 2000
 * An automatic encoding translation between Unicode and other
   encodings are implemented
 * Changes above will appear in 7.1

May 20, 2000
 * SJIS UDC (NEC selection IBM kanji) support contributed
   by Eiji Tokuya
 * Changes above will appear in 7.0.1

Mar 22, 2000
 * Add new libpq functions PQsetClientEncoding, PQclientEncoding
 * ./configure --with-mb=EUC_JP
   now deprecated. use
   ./configure --enable-multibyte=EUC_JP
   instead
   * Add SQL_ASCII regression test case
 * Add SJIS User Defined Character (UDC) support
 * All of above will appear in 7.0

July 11, 1999
 * Add support for WIN1250 (Windows Czech) as a client encoding
   (contributed by Pavel Behal)
 * fix some compiler warnings (contributed by Tomoaki Nishiyama)

Mar 23, 1999
 * Add support for KOI8(KOI8-R), WIN(CP1251), ALT(CP866)
   (thanks Oleg Broytmann for testing)
 * Fix problem with MB and locale

Jan 26, 1999
 * Add support for Big5 for frontend encoding
   (you need to create a database with EUC_TW to use Big5)
 * Add regression test case for EUC_TW
   (contributed by Jonah Kuo <jonahkuo@mail.ttn.com.tw>)

Dec 15, 1998
 * Bugs related to SQL_ASCII support fixed

Nov 5, 1998
 * 6.4 release. In this version, pg_database has "encoding"
   column that represents the database encoding

Jul 22, 1998
 * determine encoding at initdb/createdb rather than compile time
 * support for PGCLIENTENCODING when issuing COPY command
 * support for SQL92 syntax "SET NAMES"
 * support for LATIN2-5
```

```
 * add UNICODE regression test case
 * new test suite for MB
 * clean up source files

Jun 5, 1998
 * add support for the encoding translation between the backend
   and the frontend
 * new command SET CLIENT_ENCODING etc. added
 * add support for LATIN1 character set
 * enhance 8-bit cleanliness

April 21, 1998 some enhancements/fixes
 * character_length(), position(), substring() are now aware of
   multi-byte characters
 * add octet_length()
 * add --with-mb option to configure
 * new regression tests for EUC_KR
     (contributed by Soonmyung Hong)
 * add some test cases to the EUC_JP regression test
 * fix problem in regress/regress.sh in case of System V
 * fix toupper(), tolower() to handle 8bit chars

Mar 25, 1998 MB PL2 is incorporated into PostgreSQL 6.3.1

Mar 10, 1998 PL2 released
 * add regression test for EUC_JP, EUC_CN and MULE_INTERNAL
 * add an English document (this file)
 * fix problems concerning 8-bit single byte characters

Mar 1, 1998 PL1 released
```

5.2.8. WIN1250 on Windows/ODBC

The WIN1250 character set on Windows client platforms can be used with PostgreSQL with locale support enabled.

The following should be kept in mind:

- Success depends on proper system locales. This has been tested with Red Hat 6.0 and Slackware 3.6, with the `cs_CZ.iso8859-2` locale.
- Never try to set the server's database encoding to WIN1250. Always use LATIN2 instead since there is no WIN1250 locale in Unix.
- The WIN1250 encoding is usable only for Windows ODBC clients. The characters are recoded on the fly, to be displayed and stored back properly.

WIN1250 on Windows/ODBC

1. Compile PostgreSQL with locale enabled and the server-side encoding set to `LATIN2`.
2. Set up your installation. Do not forget to create locale variables in your environment. For example (this may not be correct for *your* environment):

   ```
   LC_ALL=cs_CZ.ISO8859-2
   ```

3. You have to start the server with locales set!

4. Try it with the Czech language, and have it sort on a query.

5. Install ODBC driver for PostgreSQL on your Windows machine.

6. Set up your data source properly. Include this line in your ODBC configuration dialog in the field Connect Settings:

   ```
   SET CLIENT_ENCODING = 'WIN1250';
   ```

7. Now try it again, but in Windows with ODBC.

5.3. Single-byte character set recoding

You can set up this feature with the `--enable-recode` option to `configure`. This option was formerly described as "Cyrillic recode support" which doesn't express all its power. It can be used for *any* single-byte character set recoding.

This method uses a file `charset.conf` file located in the database directory (`PGDATA`). It's a typical configuration text file where spaces and newlines separate items and records and # specifies comments. Three keywords with the following syntax are recognized here:

```
BaseCharset      server_charset
RecodeTable      from_charset to_charset file_name
HostCharset      host_spec    host_charset
```

`BaseCharset` defines the encoding of the database server. All character set names are only used for mapping inside of `charset.conf` so you can freely use typing-friendly names.

`RecodeTable` records specify translation tables between server and client. The file name is relative to the `PGDATA` directory. The table file format is very simple. There are no keywords and characters are represented by a pair of decimal or hexadecimal (0x prefixed) values on single lines:

```
char_value   translated_char_value
```

`HostCharset` records define the client character set by IP address. You can use a single IP address, an IP mask range starting from the given address or an IP interval (e.g., 127.0.0.1, 192.168.1.100/24, 192.168.1.20-192.168.1.40).

The `charset.conf` file is always processed up to the end, so you can easily specify exceptions from the previous rules. In the `src/data/` directory you will find an example `charset.conf` and a few recoding tables.

As this solution is based on the client's IP address and character set mapping there are obviously some restrictions as well. You cannot use different encodings on the same host at the same time. It is also inconvenient when you boot your client hosts into multiple operating systems. Nevertheless, when these restrictions are not limiting and you do not need multibyte characters then it is a simple and effective solution.

Chapter 6. Managing Databases

A database is a named collection of SQL objects ("database objects"). Generally, every database object (tables, functions, etc.) belongs to one and only one database. (But there are a few system catalogs, for example `pg_database`, that belong to a whole installation and are accessible from each database within the installation.) An application that connects to the database server specifies in its connection request the name of the database it wants to connect to. It is not possible to access more than one database per connection. (But an application is not restricted in the number of connections it opens to the same or other databases.)

Note

SQL calls databases "catalogs", but there is no difference in practice.

In order to create or drop databases, the PostgreSQL postmaster must be up and running (see Section 3.3, "Starting the database server").

6.1. Creating a Database

Databases are created with the query language command `CREATE DATABASE`:

```
CREATE DATABASE name
```

where *`name`* follows the usual rules for SQL identifiers. The current user automatically becomes the owner of the new database. It is the privilege of the owner of a database to remove it later on (which also removes all the objects in it, even if they have a different owner).

The creation of databases is a restricted operation. See Section 7.1.1, "User attributes" for how to grant permission.

Bootstrapping: Since you need to be connected to the database server in order to execute the `CREATE DATABASE` command, the question remains how the *first* database at any given site can be created. The first database is always created by the `initdb` command when the data storage area is initialized. (See Section 3.2, "Creating a database cluster".) By convention this database is called `template1`. So to create the first "real" database you can connect to `template1`.

The name "template1" is no accident: When a new database is created, the template database is essentially cloned. This means that any changes you make in `template1` are propagated to all subsequently created databases. This implies that you should not use the template database for real work, but when used judiciously this feature can be convenient. More details appear below.

As an extra convenience, there is also a program that you can execute from the shell to create new databases, `createdb`.

```
createdb dbname
```

`createdb` does no magic. It connects to the template1 database and issues the `CREATE DATABASE` command, exactly as described above. It uses the psql program internally. The reference page on `createdb` contains the invocation details. Note that `createdb` without any arguments will create a database with the current user name, which may or may not be what you want.

6.1.1. Template Databases

`CREATE DATABASE` actually works by copying an existing database. By default, it copies the standard system database named `template1`. Thus that database is the "template" from which new databases

are made. If you add objects to template1, these objects will be copied into subsequently created user databases. This behavior allows site-local modifications to the standard set of objects in databases. For example, if you install the procedural language plpgsql in template1, it will automatically be available in user databases without any extra action being taken when those databases are made.

There is a second standard system database named template0. This database contains the same data as the initial contents of template1, that is, only the standard objects predefined by your version of PostgreSQL. template0 should never be changed after initdb. By instructing CREATE DATABASE to copy template0 instead of template1, you can create a "virgin" user database that contains none of the site-local additions in template1. This is particularly handy when restoring a pg_dump dump: the dump script should be restored in a virgin database to ensure that one recreates the correct contents of the dumped database, without any conflicts with additions that may now be present in template1.

It is possible to create additional template databases, and indeed one might copy any database in an installation by specifying its name as the template for CREATE DATABASE. It is important to understand, however, that this is not (yet) intended as a general-purpose "COPY DATABASE" facility. In particular, it is essential that the source database be idle (no data-altering transactions in progress) for the duration of the copying operation. CREATE DATABASE will check that no backend processes (other than itself) are connected to the source database at the start of the operation, but this does not guarantee that changes cannot be made while the copy proceeds, which would result in an inconsistent copied database. Therefore, we recommend that databases used as templates be treated as read-only.

Two useful flags exist in pg_database for each database: datistemplate and datallowconn. datistemplate may be set to indicate that a database is intended as a template for CREATE DATABASE. If this flag is set, the database may be cloned by any user with CREATEDB privileges; if it is not set, only superusers and the owner of the database may clone it. If datallowconn is false, then no new connections to that database will be allowed (but existing sessions are not killed simply by setting the flag false). The template0 database is normally marked datallowconn = false to prevent modification of it. Both template0 and template1 should always be marked with datistemplate = true.

After preparing a template database, or making any changes to one, it is a good idea to perform VACUUM FREEZE or VACUUM FULL FREEZE in that database. If this is done when there are no other open transactions in the same database, then it is guaranteed that all tuples in the database are "frozen" and will not be subject to transaction ID wraparound problems. This is particularly important for a database that will have datallowconn set to false, since it will be impossible to do routine maintenance VACUUMs on such a database. See Section 8.2.3, "Preventing transaction ID wraparound failures" for more information.

Note

template1 and template0 do not have any special status beyond the fact that the name template1 is the default source database name for CREATE DATABASE and the default database-to-connect-to for various scripts such as createdb. For example, one could drop template1 and recreate it from template0 without any ill effects. This course of action might be advisable if one has carelessly added a bunch of junk in template1.

6.1.2. Alternative Locations

It is possible to create a database in a location other than the default location for the installation. Remember that all database access occurs through the database server, so any location specified must be accessible by the server.

Alternative database locations are referenced by an environment variable which gives the absolute path to the intended storage location. This environment variable must be present in the server's environment, so

it must have been defined before the server was started. (Thus, the set of available alternative locations is under the site administrator's control; ordinary users can't change it.) Any valid environment variable name may be used to reference an alternative location, although using variable names with a prefix of PGDATA is recommended to avoid confusion and conflict with other variables.

To create the variable in the environment of the server process you must first shut down the server, define the variable, initialize the data area, and finally restart the server. (See Section 3.6, “Shutting down the server” and Section 3.3, “Starting the database server”.) To set an environment variable, type

```
PGDATA2=/home/postgres/data
export PGDATA2
```

in Bourne shells, or

```
setenv PGDATA2 /home/postgres/data
```

in csh or tcsh. You have to make sure that this environment variable is always defined in the server environment, otherwise you won't be able to access that database. Therefore you probably want to set it in some sort of shell start-up file or server start-up script.

To create a data storage area in PGDATA2, ensure that the containing directory (here, /home/postgres) already exists and is writable by the user account that runs the server (see Section 3.1, “The PostgreSQL user account”). Then from the command line, type

```
initlocation PGDATA2
```

Then you can restart the server.

To create a database within the new location, use the command

```
CREATE DATABASE name WITH LOCATION = 'location'
```

where *location* is the environment variable you used, PGDATA2 in this example. The createdb command has the option -D for this purpose.

Databases created in alternative locations can be accessed and dropped like any other database.

Note

It can also be possible to specify absolute paths directly to the CREATE DATABASE command without defining environment variables. This is disallowed by default because it is a security risk. To allow it, you must compile PostgreSQL with the C preprocessor macro ALLOW_ABSOLUTE_DBPATHS defined. One way to do this is to run the compilation step like this:

```
gmake CPPFLAGS=-DALLOW_ABSOLUTE_DBPATHS all
```

6.2. Destroying a Database

Databases are destroyed with the command DROP DATABASE:

```
DROP DATABASE name
```

Only the owner of the database (i.e., the user that created it), or a superuser, can drop a database. Dropping a database removes all objects that were contained within the database. The destruction of a database cannot be undone.

You cannot execute the `DROP DATABASE` command while connected to the victim database. You can, however, be connected to any other database, including the `template1` database, which would be the only option for dropping the last user database of a given cluster.

For convenience, there is also a shell program to drop databases:

`dropdb` *`dbname`*

(Unlike `createdb`, it is not the default action to drop the database with the current user name.)

Chapter 7. Database Users and Permissions

Managing database users and their privileges is in concept similar to managing users of a Unix operating system, but the details are not identical.

7.1. Database Users

Database users are conceptually completely separate from operating system users. In practice it might be convenient to maintain a correspondence, but this is not required. Database user names are global across a database cluster installation (and not per individual database). To create a user use the `CREATE USER` SQL command:

```
CREATE USER name
```

`name` follows the rules for SQL identifiers: either unadorned without special characters, or double-quoted. To remove an existing user, use the analogous `DROP USER` command.

For convenience, the shell scripts `createuser` and `dropuser` are provided as wrappers around these SQL commands.

In order to bootstrap the database system, a freshly initialized system always contains one predefined user. This user will have the fixed id 1, and by default (unless altered when running `initdb`) it will have the same name as the operating system user that initialized the area (and is presumably being used as the user that runs the server). Customarily, this user will be named postgres. In order to create more users you first have to connect as this initial user.

The user name to use for a particular database connection is indicated by the client that is initiating the connection request in an application-specific fashion. For example, the `psql` program uses the `-U` command line option to indicate the user to connect as. The set of database users a given client connection may connect as is determined by the client authentication setup, as explained in Chapter 4, *Client Authentication*. (Thus, a client is not necessarily limited to connect as the user with the same name as its operating system user, in the same way a person is not constrained in its login name by her real name.)

7.1.1. User attributes

A database user may have a number of attributes that define its privileges and interact with the client authentication system.

superuser

A database superuser bypasses all permission checks. Also, only a superuser can create new users. To create a database superuser, use `CREATE USER name CREATEUSER`.

database creation

A user must be explicitly given permission to create databases (except for superusers, since those bypass all permission checks). To create such a user, use `CREATE USER name CREATEDB`.

password

A password is only significant if password authentication is used for client authentication. Database passwords are separate from operating system passwords. Specify a password upon user creation with `CREATE USER name PASSWORD 'string'`.

A user's attributes can be modified after creation with `ALTER USER`. See the reference pages for `CREATE USER` and `ALTER USER` for details.

7.2. Groups

As in Unix, groups are a way of logically grouping users to ease management of permissions: permissions can be granted to, or revoked from, a group as a whole. To create a group, use

```
CREATE GROUP name
```

To add users to or remove users from a group, use

```
ALTER GROUP name ADD USER uname1, ...
ALTER GROUP name DROP USER uname1, ...
```

7.3. Privileges

When a database object is created, it is assigned an owner. The owner is the user that executed the creation statement. There is currently no polished interface for changing the owner of a database object. By default, only an owner (or a superuser) can do anything with the object. In order to allow other users to use it, *privileges* must be granted.

There are several different privileges: `SELECT` (read), `INSERT` (append), `UPDATE` (write), `DELETE`, `RULE`, `REFERENCES` (foreign key), and `TRIGGER`. (See the `GRANT` manual page for more detailed information.) The right to modify or destroy an object is always the privilege of the owner only. To assign privileges, the `GRANT` command is used. So, if `joe` is an existing user, and `accounts` is an existing table, write access can be granted with

```
GRANT UPDATE ON accounts TO joe;
```

The user executing this command must be the owner of the table. To grant a privilege to a group, use

```
GRANT SELECT ON accounts TO GROUP staff;
```

The special "user" name `PUBLIC` can be used to grant a privilege to every user on the system. Writing `ALL` in place of a specific privilege specifies that all privileges will be granted.

To revoke a privilege, use the fittingly named `REVOKE` command:

```
REVOKE ALL ON accounts FROM PUBLIC;
```

The special privileges of the table owner (i.e., the right to do `DROP`, `GRANT`, `REVOKE`, etc) are always implicit in being the owner, and cannot be granted or revoked. But the table owner can choose to revoke his own ordinary privileges, for example to make a table read-only for himself as well as others.

7.4. Functions and Triggers

Functions and triggers allow users to insert code into the backend server that other users may execute without knowing it. Hence, both mechanisms permit users to *Trojan horse* others with relative impunity. The only real protection is tight control over who can define functions (e.g., write to relations with SQL fields) and triggers. Audit trails and alerters on the system catalogs `pg_class`, `pg_shadow` and `pg_group` are also possible.

Functions written in any language except SQL run inside the backend server process with the operating systems permissions of the database server daemon process. It is possible to change the server's internal

data structures from inside of trusted functions. Hence, among many other things, such functions can circumvent any system access controls. This is an inherent problem with user-defined C functions.

Chapter 8. Routine Database Maintenance Tasks

8.1. General Discussion

There are a few routine maintenance chores that must be performed on a regular basis to keep a PostgreSQL installation running smoothly. The tasks discussed here are repetitive in nature and can easily be automated using standard Unix tools such as `cron` scripts. But it is the database administrator's responsibility to set up appropriate scripts, and to check that they execute successfully.

One obvious maintenance task is creation of backup copies of the data on a regular schedule. Without a recent backup, you have no chance of recovery after a catastrophe (disk failure, fire, mistakenly dropping a critical table, etc). The backup and recovery mechanisms available in PostgreSQL are discussed at length in Chapter 9, *Backup and Restore*.

The other main category of maintenance task is periodic "vacuuming" of the database. This activity is discussed in Section 8.2, "Routine Vacuuming".

Something else that might need periodic attention is log file management. This is discussed in Section 8.3, "Log File Maintenance".

PostgreSQL is low-maintenance compared to some other database products. Nonetheless, appropriate attention to these tasks will go far towards ensuring a pleasant and productive experience with the system.

8.2. Routine Vacuuming

PostgreSQL's `VACUUM` command must be run on a regular basis for several reasons:

1. To recover disk space occupied by updated or deleted rows.

2. To update data statistics used by the PostgreSQL query planner.

3. To protect against loss of very old data due to *transaction ID wraparound*.

The frequency and scope of `VACUUM`s performed for each of these reasons will vary depending on the needs of each installation. Therefore, database administrators must understand these issues and develop an appropriate maintenance strategy. This section concentrates on explaining the high-level issues; for details about command syntax and so on, see the `VACUUM` command reference page.

Beginning in PostgreSQL 7.2, the standard form of `VACUUM` can run in parallel with normal database operations (selects, inserts, updates, deletes, but not changes to table schemas). Routine vacuuming is therefore not nearly as intrusive as it was in prior releases, and it's not as critical to try to schedule it at low-usage times of day.

8.2.1. Recovering disk space

In normal PostgreSQL operation, an `UPDATE` or `DELETE` of a row does not immediately remove the old *tuple* (version of the row). This approach is necessary to gain the benefits of multiversion concurrency control (see the *User's Guide*): the tuple must not be deleted while it is still potentially visible to other transactions. But eventually, an outdated or deleted tuple is no longer of interest to any transaction. The

space it occupies must be reclaimed for reuse by new tuples, to avoid infinite growth of disk space requirements. This is done by running VACUUM.

Clearly, a table that receives frequent updates or deletes will need to be vacuumed more often than tables that are seldom updated. It may be useful to set up periodic cron tasks that vacuum only selected tables, skipping tables that are known not to change often. This is only likely to be helpful if you have both large heavily-updated tables and large seldom-updated tables --- the extra cost of vacuuming a small table isn't enough to be worth worrying about.

The standard form of VACUUM is best used with the goal of maintaining a fairly level steady-state usage of disk space. The standard form finds old tuples and makes their space available for re-use within the table, but it does not try very hard to shorten the table file and return disk space to the operating system. If you need to return disk space to the operating system you can use VACUUM FULL --- but what's the point of releasing disk space that will only have to be allocated again soon? Moderately frequent standard VACUUMs are a better approach than infrequent VACUUM FULLs for maintaining heavily-updated tables.

Recommended practice for most sites is to schedule a database-wide VACUUM once a day at a low-usage time of day, supplemented by more frequent vacuuming of heavily-updated tables if necessary. (If you have multiple databases in an installation, don't forget to vacuum each one; the vacuumdb script may be helpful.) Use plain VACUUM, not VACUUM FULL, for routine vacuuming for space recovery.

VACUUM FULL is recommended for cases where you know you have deleted the majority of tuples in a table, so that the steady-state size of the table can be shrunk substantially with VACUUM FULL's more aggressive approach.

If you have a table whose contents are deleted completely every so often, consider doing it with TRUNCATE rather than using DELETE followed by VACUUM.

8.2.2. Updating planner statistics

The PostgreSQL query planner relies on statistical information about the contents of tables in order to generate good plans for queries. These statistics are gathered by the ANALYZE command, which can be invoked by itself or as an optional step in VACUUM. It is important to have reasonably accurate statistics, otherwise poor choices of plans may degrade database performance.

As with vacuuming for space recovery, frequent updates of statistics are more useful for heavily-updated tables than for seldom-updated ones. But even for a heavily-updated table, there may be no need for statistics updates if the statistical distribution of the data is not changing much. A simple rule of thumb is to think about how much the minimum and maximum values of the columns in the table change. For example, a timestamp column that contains the time of row update will have a constantly-increasing maximum value as rows are added and updated; such a column will probably need more frequent statistics updates than, say, a column containing URLs for pages accessed on a website. The URL column may receive changes just as often, but the statistical distribution of its values probably changes relatively slowly.

It is possible to run ANALYZE on specific tables and even just specific columns of a table, so the flexibility exists to update some statistics more frequently than others if your application requires it. In practice, however, the usefulness of this feature is doubtful. Beginning in PostgreSQL 7.2, ANALYZE is a fairly fast operation even on large tables, because it uses a statistical random sampling of the rows of a table rather than reading every single row. So it's probably much simpler to just run it over the whole database every so often.

Tip

Although per-column tweaking of ANALYZE frequency may not be very productive, you may well find it worthwhile to do per-column adjustment of the level of detail of the statistics collected

by ANALYZE. Columns that are heavily used in WHERE clauses and have highly irregular data distributions may require a finer-grain data histogram than other columns. See ALTER TABLE SET STATISTICS.

Recommended practice for most sites is to schedule a database-wide ANALYZE once a day at a low-usage time of day; this can usefully be combined with a nightly VACUUM. However, sites with relatively slowly changing table statistics may find that this is overkill, and that less-frequent ANALYZE runs are sufficient.

8.2.3. Preventing transaction ID wraparound failures

PostgreSQL's MVCC transaction semantics depend on being able to compare transaction ID (*XID*) numbers: a tuple with an insertion XID newer than the current transaction's XID is "in the future" and should not be visible to the current transaction. But since transaction IDs have limited size (32 bits at this writing) an installation that runs for a long time (more than 4 billion transactions) will suffer *transaction ID wraparound*: the XID counter wraps around to zero, and all of a sudden transactions that were in the past appear to be in the future --- which means their outputs become invisible. In short, catastrophic data loss. (Actually the data is still there, but that's cold comfort if you can't get at it.)

Prior to PostgreSQL 7.2, the only defense against XID wraparound was to re-initdb at least every 4 billion transactions. This of course was not very satisfactory for high-traffic sites, so a better solution has been devised. The new approach allows an installation to remain up indefinitely, without initdb or any sort of restart. The price is this maintenance requirement: *every table in the database must be vacuumed at least once every billion transactions*.

In practice this isn't an onerous requirement, but since the consequences of failing to meet it can be complete data loss (not just wasted disk space or slow performance), some special provisions have been made to help database administrators keep track of the time since the last VACUUM. The remainder of this section gives the details.

The new approach to XID comparison distinguishes two special XIDs, numbers 1 and 2 (BootstrapXID and FrozenXID). These two XIDs are always considered older than every normal XID. Normal XIDs (those greater than 2) are compared using modulo-2^{31} arithmetic. This means that for every normal XID, there are two billion XIDs that are "older" and two billion that are "newer"; another way to say it is that the normal XID space is circular with no endpoint. Therefore, once a tuple has been created with a particular normal XID, the tuple will appear to be "in the past" for the next two billion transactions, no matter which normal XID we are talking about. If the tuple still exists after more than two billion transactions, it will suddenly appear to be in the future. To prevent data loss, old tuples must be reassigned the XID FrozenXID sometime before they reach the two-billion-transactions-old mark. Once they are assigned this special XID, they will appear to be "in the past" to all normal transactions regardless of wraparound issues, and so such tuples will be good until deleted, no matter how long that is. This reassignment of XID is handled by VACUUM.

VACUUM's normal policy is to reassign FrozenXID to any tuple with a normal XID more than one billion transactions in the past. This policy preserves the original insertion XID until it is not likely to be of interest anymore (in fact, most tuples will probably live and die without ever being "frozen"). With this policy, the maximum safe interval between VACUUMs of any table is exactly one billion transactions: if you wait longer, it's possible that a tuple that was not quite old enough to be reassigned last time is now more than two billion transactions old and has wrapped around into the future --- ie, is lost to you. (Of course, it'll reappear after another two billion transactions, but that's no help.)

Since periodic VACUUMs are needed anyway for the reasons described earlier, it's unlikely that any table would not be vacuumed for as long as a billion transactions. But to help administrators ensure this constraint is met, VACUUM stores transaction ID statistics in the system table pg_database. In particular, the datfrozenxid field of a database's pg_database row is updated at the completion of any database-wide vacuum operation (ie, VACUUM that does not name a specific table). The value stored in this field

is the freeze cutoff XID that was used by that VACUUM command. All normal XIDs older than this cutoff XID are guaranteed to have been replaced by FrozenXID within that database. A convenient way to examine this information is to execute the query

```
SELECT datname, age(datfrozenxid) FROM pg_database;
```

The age column measures the number of transactions from the cutoff XID to the current transaction's XID.

With the standard freezing policy, the age column will start at one billion for a freshly-vacuumed database. When the age approaches two billion, the database must be vacuumed again to avoid risk of wraparound failures. Recommended practice is to vacuum each database at least once every half-a-billion (500 million) transactions, so as to provide plenty of safety margin. To help meet this rule, each database-wide VACUUM automatically delivers a warning if there are any pg_database entries showing an age of more than 1.5 billion transactions, for example:

```
play=# vacuum;
NOTICE:  Some databases have not been vacuumed in 1613770184
 transactions.
        Better vacuum them within 533713463 transactions,
        or you may have a wraparound failure.
VACUUM
```

VACUUM with the FREEZE option uses a more aggressive freezing policy: tuples are frozen if they are old enough to be considered good by all open transactions. In particular, if a VACUUM FREEZE is performed in an otherwise-idle database, it is guaranteed that *all* tuples in that database will be frozen. Hence, as long as the database is not modified in any way, it will not need subsequent vacuuming to avoid transaction ID wraparound problems. This technique is used by initdb to prepare the template0 database. It should also be used to prepare any user-created databases that are to be marked datallowconn = false in pg_database, since there isn't any convenient way to vacuum a database that you can't connect to. Note that VACUUM's automatic warning message about unvacuumed databases will ignore pg_database entries with datallowconn = false, so as to avoid giving false warnings about these databases; therefore it's up to you to ensure that such databases are frozen correctly.

8.3. Log File Maintenance

It's a good idea to save the database server's log output somewhere, rather than just routing it to /dev/null. The log output is invaluable when it comes time to diagnose problems. However, the log output tends to be voluminous (especially at higher debug levels) and you won't want to save it indefinitely. You need to “rotate” the log files so that new log files are started and old ones thrown away every so often.

If you simply direct the postmaster's stderr into a file, the only way to truncate the log file is to stop and restart the postmaster. This may be OK for development setups but you won't want to run a production server that way.

The simplest production-grade approach to managing log output is to send it all to syslog and let syslog deal with file rotation. To do this, make sure PostgreSQL was built with the --enable-syslog configure option, and set syslog to 2 (log to syslog only) in postgresql.conf. Then you can send a SIGHUP signal to the syslog daemon whenever you want to force it to start writing a new log file.

On many systems, however, syslog is not very reliable, particularly with large log messages; it may truncate or drop messages just when you need them the most. You may find it more useful to pipe the postmaster's stderr to some type of log rotation script. If you start the postmaster with pg_ctl, then the postmaster's stderr is already redirected to stdout, so you just need a pipe command:

```
pg_ctl start | logrotate
```

The PostgreSQL distribution doesn't include a suitable log rotation program, but there are many available on the net; one is included in the Apache distribution, for example.

Chapter 9. Backup and Restore

As everything that contains valuable data, PostgreSQL databases should be backed up regularly. While the procedure is essentially simple, it is important to have a basic understanding of the underlying techniques and assumptions.

There are two fundamentally different approaches to backing up PostgreSQL data:

- SQL dump
- File system level backup

9.1. SQL Dump

The idea behind the SQL-dump method is to generate a text file with SQL commands that, when fed back to the server, will recreate the database in the same state as it was at the time of the dump. PostgreSQL provides the utility program pg_dump for this purpose. The basic usage of this command is:

```
pg_dump dbname > outfile
```

As you see, pg_dump writes its results to the standard output. We will see below how this can be useful.

pg_dump is a regular PostgreSQL client application (albeit a particularly clever one). This means that you can do this backup procedure from any remote host that has access to the database. But remember that pg_dump does not operate with special permissions. In particular, you must have read access to all tables that you want to back up, so in practice you almost always have to be a database superuser.

To specify which database server pg_dump should contact, use the command line options `-h host` and `-p port`. The default host is the local host or whatever your `PGHOST` environment variable specifies. Similarly, the default port is indicated by the `PGPORT` environment variable or, failing that, by the compiled-in default. (Conveniently, the server will normally have the same compiled-in default.)

As any other PostgreSQL client application, pg_dump will by default connect with the database user name that is equal to the current Unix user name. To override this, either specify the `-U` option or set the environment variable `PGUSER`. Remember that pg_dump connections are subject to the normal client authentication mechanisms (which are described in Chapter 4, *Client Authentication*).

Dumps created by pg_dump are internally consistent, that is, updates to the database while pg_dump is running will not be in the dump. pg_dump does not block other operations on the database while it is working. (Exceptions are those operations that need to operate with an exclusive lock, such as `VACUUM FULL`.)

Important

When your database schema relies on OIDs (for instance as foreign keys) you must instruct pg_dump to dump the OIDs as well. To do this, use the `-o` command line option. “Large objects” are not dumped by default, either. See pg_dump's command reference page if you use large objects.

9.1.1. Restoring the dump

The text files created by pg_dump are intended to be read in by the psql program. The general command form to restore a dump is

```
psql dbname < infile
```

where *infile* is what you used as *outfile* for the pg_dump command. The database *dbname* will not be created by this command, you must create it yourself from template0 before executing psql (e.g., with `createdb -T template0 dbname`). psql supports similar options to pg_dump for controlling the database server location and the user names. See its reference page for more information.

If the objects in the original database were owned by different users, then the dump will instruct psql to connect as each affected user in turn and then create the relevant objects. This way the original ownership is preserved. This also means, however, that all these users must already exist, and furthermore that you must be allowed to connect as each of them. It might therefore be necessary to temporarily relax the client authentication settings.

The ability of pg_dump and psql to write to or read from pipes makes it possible to dump a database directly from one server to another, for example

```
pg_dump -h host1 dbname | psql -h host2 dbname
```

Important

The dumps produced by pg_dump are relative to template0. This means that any languages, procedures, etc. added to template1 will also be dumped by pg_dump. As a result, when restoring, if you are using a customized template1, you must create the empty database from template0, as in the example above.

9.1.2. Using `pg_dumpall`

The above mechanism is cumbersome and inappropriate when backing up an entire database cluster. For this reason the pg_dumpall program is provided. pg_dumpall backs up each database in a given cluster and also makes sure that the state of global data such as users and groups is preserved. The call sequence for pg_dumpall is simply

```
pg_dumpall > outfile
```

The resulting dumps can be restored with psql as described above. But in this case it is definitely necessary that you have database superuser access, as that is required to restore the user and group information.

9.1.3. Large Databases

Acknowledgement

Originally written by Hannu Krosing (<hannu@trust.ee>) on 1999-06-19

Since PostgreSQL allows tables larger than the maximum file size on your system, it can be problematic to dump the table to a file, since the resulting file will likely be larger than the maximum size allowed by your system. As pg_dump writes to the standard output, you can just use standard *nix tools to work around this possible problem.

Use compressed dumps. Use your favorite compression program, for example gzip.

```
pg_dump dbname | gzip > filename.gz
```

Reload with

```
createdb dbname
gunzip -c filename.gz | psql dbname
```

or

```
cat filename.gz | gunzip | psql dbname
```

Use split. This allows you to split the output into pieces that are acceptable in size to the underlying file system. For example, to make chunks of 1 megabyte:

```
pg_dump dbname | split -b 1m - filename
```

Reload with

```
createdb dbname
cat filename* | psql dbname
```

Use the custom dump format. If PostgreSQL was built on a system with the zlib compression library installed, the custom dump format will compress data as it writes it to the output file. For large databases, this will produce similar dump sizes to using `gzip`, but has the added advantage that the tables can be restored selectively. The following command dumps a database using the custom dump format:

```
pg_dump -Fc dbname > filename
```

See the pg_dump and pg_restore reference pages for details.

9.1.4. Caveats

pg_dump (and by implication pg_dumpall) has a few limitations which stem from the difficulty to reconstruct certain information from the system catalogs.

Specifically, the order in which pg_dump writes the objects is not very sophisticated. This can lead to problems for example when functions are used as column default values. The only answer is to manually reorder the dump. If you created circular dependencies in your schema then you will have more work to do.

For reasons of backward compatibility, pg_dump does not dump large objects by default. To dump large objects you must use either the custom or the TAR output format, and use the -b option in pg_dump. See the reference pages for details. The directory `contrib/pg_dumplo` of the PostgreSQL source tree also contains a program that can dump large objects.

Please familiarize yourself with the pg_dump reference page.

9.2. File system level backup

An alternative backup strategy is to directly copy the files that PostgreSQL uses to store the data in the database. In Section 3.2, “Creating a database cluster” it is explained where these files are located, but you have probably found them already if you are interested in this method. You can use whatever method you prefer for doing usual file system backups, for example

```
tar -cf backup.tar /usr/local/pgsql/data
```

There are two restrictions, however, which make this method impractical, or at least inferior to the pg_dump method:

1. The database server *must* be shut down in order to get a usable backup. Half-way measures such as disallowing all connections will not work as there is always some buffering going on. For this reason it is also not advisable to trust file systems that claim to support “consistent snapshots”. Information about stopping the server can be found in Section 3.6, “Shutting down the server”.

   Needless to say that you also need to shut down the server before restoring the data.

2. If you have dug into the details of the file system layout you may be tempted to try to back up or restore only certain individual tables or databases from their respective files or directories. This will *not* work because the information contained in these files contains only half the truth. The other half is in the commit log files `pg_clog/*`, which contain the commit status of all transactions. A table file is only usable with this information. Of course it is also impossible to restore only a table and the associated `pg_clog` data because that will render all other tables in the database cluster useless.

Also note that the file system backup will not necessarily be smaller than an SQL dump. On the contrary, it will most likely be larger. (pg_dump does not need to dump the contents of indexes for example, just the commands to recreate them.)

9.3. Migration between releases

As a general rule, the internal data storage format is subject to change between releases of PostgreSQL. This does not apply to different "patch levels", these always have compatible storage formats. For example, releases 7.0.1, 7.1.2, and 7.2 are not compatible, whereas 7.1.1 and 7.1.2 are. When you update between compatible versions, then you can simply reuse the data area in disk by the new executables. Otherwise you need to "back up" your data and "restore" it on the new server, using pg_dump. (There are checks in place that prevent you from doing the wrong thing, so no harm can be done by confusing these things.) The precise installation procedure is not subject of this section, these details are in Chapter 1, *Installation Instructions*.

The least downtime can be achieved by installing the new server in a different directory and running both the old and the new servers in parallel, on different ports. Then you can use something like

```
pg_dumpall -p 5432 | psql -d template1 -p 6543
```

to transfer your data, or use an intermediate file if you want. Then you can shut down the old server and start the new server at the port the old one was running at. You should make sure that the database is not updated after you run pg_dumpall, otherwise you will obviously lose that data. See Chapter 4, *Client Authentication* for information on how to prohibit access. In practice you probably want to test your client applications on the new setup before switching over.

If you cannot or do not want to run two servers in parallel you can do the back up step before installing the new version, bring down the server, move the old version out of the way, install the new version, start the new server, restore the data. For example:

```
pg_dumpall > backup
pg_ctl stop
mv /usr/local/pgsql /usr/local/pgsql.old
cd /usr/src/postgresql-7.2.8
gmake install
initdb -D /usr/local/pgsql/data
postmaster -D /usr/local/pgsql/data
psql < backup
```

See Chapter 3, *Server Runtime Environment* about ways to start and stop the server and other details. The installation instructions will advise you of strategic places to perform these steps.

Note

When you "move the old installation out of the way" it is no longer perfectly usable. Some parts of the installation contain information about where the other parts are located. This is usually not a big problem but if you plan on using two installations in parallel for a while you should assign them different installation directories at build time.

Chapter 10. Monitoring Database Activity

A database administrator frequently wonders “what is the system doing right now?” This chapter discusses how to find that out.

Several tools are available for monitoring database activity and analyzing performance. Most of this chapter is devoted to describing PostgreSQL's *statistics collector*, but one should not neglect regular Unix monitoring programs such as `ps` and `top`. Also, once one has identified a poorly-performing query, further investigation may be needed using PostgreSQL's `EXPLAIN` command. The *User's Guide* discusses `EXPLAIN` and other methods for understanding the behavior of an individual query.

10.1. Standard Unix Tools

On most platforms, PostgreSQL modifies its command title as reported by `ps`, so that individual server processes can readily be identified. A sample display is

```
$ 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
```

(The appropriate invocation of `ps` varies across different platforms, as do the details of what is shown. This example is from a recent Linux system.) The first process listed here is the *postmaster*, the master server process. The command arguments shown for it are the same ones given when it was launched. The next two processes implement the statistics collector, which will be described in detail in the next section. (These will not be present if you have set the system not to start the statistics collector.) Each of the remaining processes is a server process handling one client connection. Each such process sets its command line display in the form

`postgres: `*`user database host activity`*

The user, database, and connection source host items remain the same for the life of the client connection, but the activity indicator changes. The activity may be `idle` (ie, waiting for a client command), `idle in transaction` (waiting for client inside a BEGIN block), or a command type name such as `SELECT`. Also, `waiting` is attached if the server is presently waiting on a lock held by another server process. In the above example we can infer that process 1003 is waiting for process 1016 to complete its transaction and thereby release some lock or other.

Tip

Solaris requires special handling. You must use `/usr/ucb/ps`, rather than `/bin/ps`. You also must use two `w` flags, not just one. In addition, your original invocation of the postmaster must

have a shorter `ps` status display than that provided by each backend. If you fail to do all three things, the `ps` output for each backend will be the original postmaster command line.

10.2. Statistics Collector

PostgreSQL's *statistics collector* is a subsystem that supports collection and reporting of information about server activity. Presently, the collector can count accesses to tables and indexes in both disk-block and individual-row terms. It also supports determining the exact query currently being executed by other server processes.

10.2.1. Statistics Collection Configuration

Since collection of statistics adds some overhead to query execution, the system can be configured to collect or not collect information. This is controlled by configuration variables that are normally set in `postgresql.conf` (see Section 3.4, “Run-time configuration” for details about setting configuration variables).

The variable `STATS_START_COLLECTOR` must be set to `true` for the statistics collector to be launched at all. This is the default and recommended setting, but it may be turned off if you have no interest in statistics and want to squeeze out every last drop of overhead. (The savings is likely to be small, however.) Note that this option cannot be changed while the server is running.

The variables `STATS_COMMAND_STRING`, `STATS_BLOCK_LEVEL`, and `STATS_ROW_LEVEL` control how much information is actually sent to the collector, and thus determine how much runtime overhead occurs. These respectively determine whether a server process sends its current command string, disk-block-level access statistics, and row-level access statistics to the collector. Normally these variables are set in `postgresql.conf` so that they apply to all server processes, but it is possible to turn them on or off in individual server processes using the `SET` command. (To prevent ordinary users from hiding their activity from the administrator, only superusers are allowed to change these variables with `SET`.)

Important

Since the variables `STATS_COMMAND_STRING`, `STATS_BLOCK_LEVEL`, and `STATS_ROW_LEVEL` default to `false`, no statistics are actually collected in the default configuration! You must turn one or more of them on before you will get useful results from the statistical display functions.

10.2.2. Viewing Collected Statistics

Several predefined views are available to show the results of statistics collection. Alternatively, one can build custom views using the underlying statistics functions.

When using the statistics to monitor current activity, it is important to realize that the information does not update instantaneously. Each individual server process transmits new access counts to the collector just before waiting for another client command; so a query still in progress does not affect the displayed totals. Also, the collector itself emits new totals at most once per PGSTAT_STAT_INTERVAL (500 milliseconds by default). So the displayed totals lag behind actual activity.

Another important point is that when a server process is asked to display any of these statistics, it first fetches the most recent totals emitted by the collector process. It then continues to use this snapshot for all statistical views and functions until the end of its current transaction. So the statistics will appear not to change as long as you continue the current transaction. This is a feature, not a bug, because it allows you to perform several queries on the statistics and correlate the results without worrying that the numbers are changing underneath you. But if you want to see new results with each query, be sure to do the queries outside any transaction block.

Table 10.1. Standard Statistics Views

View Name	Description
pg_stat_activity	One row per server process, showing process PID, database, user, and current query. The current query column is only available to superusers; for others it reads as NULL. (Note that because of the collector's reporting delay, current query will only be up-to-date for long-running queries.)
pg_stat_database	One row per database, showing number of active backends, total transactions committed and total rolled back in that database, total disk blocks read, and total number of buffer hits (ie, block read requests avoided by finding the block already in buffer cache).
pg_stat_all_tables	For each table in the current database, total numbers of sequential and index scans, total numbers of tuples returned by each type of scan, and totals of tuple insertions, updates, and deletes.
pg_stat_sys_tables	Same as pg_stat_all_tables, except that only system tables are shown.
pg_stat_user_tables	Same as pg_stat_all_tables, except that only user tables are shown.
pg_stat_all_indexes	For each index in the current database, the total number of index scans that have used that index, the number of index tuples read, and the number of successfully fetched heap tuples (this may be less when there are index entries pointing to expired heap tuples).
pg_stat_sys_indexes	Same as pg_stat_all_indexes, except that only indexes on system tables are shown.
pg_stat_user_indexes	Same as pg_stat_all_indexes, except that only indexes on user tables are shown.
pg_statio_all_tables	For each table in the current database, the total number of disk blocks read from that table, the number of buffer hits, the numbers of disk blocks read and buffer hits in all the indexes of that table, the numbers of disk blocks read and buffer hits from the table's auxiliary TOAST table (if any), and the numbers of disk blocks read and buffer hits for the TOAST table's index.
pg_statio_sys_tables	Same as pg_statio_all_tables, except that only system tables are shown.
pg_statio_user_tables	Same as pg_statio_all_tables, except that only user tables are shown.
pg_statio_all_indexes	For each index in the current database, the numbers of disk blocks read and buffer hits in that index.
pg_statio_sys_indexes	Same as pg_statio_all_indexes, except that only indexes on system tables are shown.

View Name	Description
pg_statio_user_indexes	Same as pg_statio_all_indexes, except that only indexes on user tables are shown.
pg_statio_all_sequences	For each sequence object in the current database, the numbers of disk blocks read and buffer hits in that sequence.
pg_statio_sys_sequences	Same as pg_statio_all_sequences, except that only system sequences are shown. (Presently, no system sequences are defined, so this view is always empty.)
pg_statio_user_sequences	Same as pg_statio_all_sequences, except that only user sequences are shown.

The per-index statistics are particularly useful to determine which indexes are being used and how effective they are.

The pg_statio_ views are primarily useful to determine the effectiveness of the buffer cache. When the number of actual disk reads is much smaller than the number of buffer hits, then the cache is satisfying most read requests without invoking a kernel call.

Other ways of looking at the statistics can be set up by writing queries that use the same underlying statistics access functions as these standard views do. The per-database access functions accept a database OID to identify which database to report on. The per-table and per-index functions accept a table or index OID (note that only tables and indexes in the current database can be seen with these functions). The per-backend access functions accept a backend ID number, which ranges from one to the number of currently active backends.

Table 10.2. Statistics Access Functions

Function	Return Type	Description
pg_stat_get_db_numbackends(oid)	integer	Number of active backends in database
pg_stat_get_db_xact_commit(oid)	bigint	Transactions committed in database
pg_stat_get_db_xact_rollback(oid)	bigint	Transactions rolled back in database
pg_stat_get_db_blocks_fetched(oid)	bigint	Number of disk block fetch requests for database
pg_stat_get_db_blocks_hit(oid)	bigint	Number of disk block requests found in cache for database
pg_stat_get_numscans(oid)	bigint	Number of sequential scans done when argument is a table, or number of index scans done when argument is an index
pg_stat_get_tuples_returned(oid)	bigint	Number of tuples read by sequential scans when argument is a table, or number of index tuples read when argument is an index
pg_stat_get_tuples_fetched(oid)	bigint	Number of valid (unexpired) table tuples fetched by sequential scans

Function	Return Type	Description
		when argument is a table, or fetched by index scans using this index when argument is an index
pg_stat_get_tuples_inserted(oid)	bigint	Number of tuples inserted into table
pg_stat_get_tuples_updated(oid)	bigint	Number of tuples updated in table
pg_stat_get_tuples_deleted(oid)	bigint	Number of tuples deleted from table
pg_stat_get_blocks_fetched(oid)	bigint	Number of disk block fetch requests for table or index
pg_stat_get_blocks_hit(oid)	bigint	Number of disk block requests found in cache for table or index
pg_stat_get_backend_idset()	set of integer	Set of currently active backend IDs (from 1 to N where N is the number of active backends). See usage example below.
pg_stat_get_backend_pid(integer)	integer	PID of backend process
pg_stat_get_backend_dbid(integer)	oid	Database ID of backend process
pg_stat_get_backend_userid(integer)	oid	User ID of backend process
pg_stat_get_backend_activity(integer)	text	Current query of backend process (NULL if caller is not superuser)

Note: blocks_fetched minus blocks_hit gives the number of kernel read() calls issued for the table, index, or database; but the actual number of physical reads is usually lower due to kernel-level buffering.

The function pg_stat_get_backend_idset provides a convenient way to generate one row for each active backend. For example, to show the PIDs and current queries of all backends:

```
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;
```

Chapter 11. Write-Ahead Logging (WAL)

Author

Vadim Mikheev and Oliver Elphick

11.1. General Description

Write Ahead Logging (WAL) is a standard approach to transaction logging. Its detailed description may be found in most (if not all) books about transaction processing. Briefly, WAL's central concept is that changes to data files (where tables and indexes reside) must be written only after those changes have been logged - that is, when log records have been flushed to permanent storage. When we follow this procedure, we do not need to flush data pages to disk on every transaction commit, because we know that in the event of a crash we will be able to recover the database using the log: any changes that have not been applied to the data pages will first be redone from the log records (this is roll-forward recovery, also known as REDO) and then changes made by uncommitted transactions will be removed from the data pages (roll-backward recovery - UNDO).

11.1.1. Immediate Benefits of WAL

The first obvious benefit of using WAL is a significantly reduced number of disk writes, since only the log file needs to be flushed to disk at the time of transaction commit; in multiuser environments, commits of many transactions may be accomplished with a single `fsync()` of the log file. Furthermore, the log file is written sequentially, and so the cost of syncing the log is much less than the cost of flushing the data pages.

The next benefit is consistency of the data pages. The truth is that, before WAL, PostgreSQL was never able to guarantee consistency in the case of a crash. Before WAL, any crash during writing could result in:

1. index tuples pointing to nonexistent table rows

2. index tuples lost in split operations

3. totally corrupted table or index page content, because of partially written data pages

Problems with indexes (problems 1 and 2) could possibly have been fixed by additional `fsync()` calls, but it is not obvious how to handle the last case without WAL; WAL saves the entire data page content in the log if that is required to ensure page consistency for after-crash recovery.

11.1.2. Future Benefits

In this first release of WAL, UNDO operation is not implemented, because of lack of time. This means that changes made by aborted transactions will still occupy disk space and that we still need a permanent `pg_clog` file to hold the status of transactions, since we are not able to re-use transaction identifiers. Once UNDO is implemented, `pg_clog` will no longer be required to be permanent; it will be possible to remove `pg_clog` at shutdown. (However, the urgency of this concern has decreased greatly with the adoption of a segmented storage method for `pg_clog` --- it is no longer necessary to keep old `pg_clog` entries around forever.)

With UNDO, it will also be possible to implement *savepoints* to allow partial rollback of invalid transaction operations (parser errors caused by mistyping commands, insertion of duplicate primary/unique keys and

so on) with the ability to continue or commit valid operations made by the transaction before the error. At present, any error will invalidate the whole transaction and require a transaction abort.

WAL offers the opportunity for a new method for database on-line backup and restore (BAR). To use this method, one would have to make periodic saves of data files to another disk, a tape or another host and also archive the WAL log files. The database file copy and the archived log files could be used to restore just as if one were restoring after a crash. Each time a new database file copy was made the old log files could be removed. Implementing this facility will require the logging of data file and index creation and deletion; it will also require development of a method for copying the data files (operating system copy commands are not suitable).

A difficulty standing in the way of realizing these benefits is that they require saving WAL entries for considerable periods of time (eg, as long as the longest possible transaction if transaction UNDO is wanted). The present WAL format is extremely bulky since it includes many disk page snapshots. This is not a serious concern at present, since the entries only need to be kept for one or two checkpoint intervals; but to achieve these future benefits some sort of compressed WAL format will be needed.

11.2. Implementation

WAL is automatically enabled from release 7.1 onwards. No action is required from the administrator with the exception of ensuring that the additional disk-space requirements of the WAL logs are met, and that any necessary tuning is done (see Section 11.3, “WAL Configuration”).

WAL logs are stored in the directory *$PGDATA*/pg_xlog, as a set of segment files, each 16 MB in size. Each segment is divided into 8 kB pages. The log record headers are described in access/xlog.h; record content is dependent on the type of event that is being logged. Segment files are given ever-increasing numbers as names, starting at 0000000000000000. The numbers do not wrap, at present, but it should take a very long time to exhaust the available stock of numbers.

The WAL buffers and control structure are in shared memory, and are handled by the backends; they are protected by lightweight locks. The demand on shared memory is dependent on the number of buffers. The default size of the WAL buffers is 8 buffers of 8 kB each, or 64 kB total.

It is of advantage if the log is located on another disk than the main database files. This may be achieved by moving the directory, pg_xlog, to another location (while the postmaster is shut down, of course) and creating a symbolic link from the original location in *$PGDATA* to the new location.

The aim of WAL, to ensure that the log is written before database records are altered, may be subverted by disk drives that falsely report a successful write to the kernel, when, in fact, they have only cached the data and not yet stored it on the disk. A power failure in such a situation may still lead to irrecoverable data corruption. Administrators should try to ensure that disks holding PostgreSQL's log files do not make such false reports.

11.2.1. Database Recovery with WAL

After a checkpoint has been made and the log flushed, the checkpoint's position is saved in the file pg_control. Therefore, when recovery is to be done, the backend first reads pg_control and then the checkpoint record; then it performs the REDO operation by scanning forward from the log position indicated in the checkpoint record. Because the entire content of data pages is saved in the log on the first page modification after a checkpoint, all pages changed since the checkpoint will be restored to a consistent state.

Using pg_control to get the checkpoint position speeds up the recovery process, but to handle possible corruption of pg_control, we should actually implement the reading of existing log segments in reverse order -- newest to oldest -- in order to find the last checkpoint. This has not been implemented, yet.

11.3. WAL Configuration

There are several WAL-related parameters that affect database performance. This section explains their use. Consult Section 3.4, “Run-time configuration” for details about setting configuration parameters.

There are two commonly used WAL functions: `LogInsert` and `LogFlush`. `LogInsert` is used to place a new record into the WAL buffers in shared memory. If there is no space for the new record, `LogInsert` will have to write (move to kernel cache) a few filled WAL buffers. This is undesirable because `LogInsert` is used on every database low level modification (for example, tuple insertion) at a time when an exclusive lock is held on affected data pages, so the operation needs to be as fast as possible. What is worse, writing WAL buffers may also force the creation of a new log segment, which takes even more time. Normally, WAL buffers should be written and flushed by a `LogFlush` request, which is made, for the most part, at transaction commit time to ensure that transaction records are flushed to permanent storage. On systems with high log output, `LogFlush` requests may not occur often enough to prevent WAL buffers being written by `LogInsert`. On such systems one should increase the number of WAL buffers by modifying the `postgresql.conf WAL_BUFFERS` parameter. The default number of WAL buffers is 8. Increasing this value will correspondingly increase shared memory usage.

Checkpoints are points in the sequence of transactions at which it is guaranteed that the data files have been updated with all information logged before the checkpoint. At checkpoint time, all dirty data pages are flushed to disk and a special checkpoint record is written to the log file. As result, in the event of a crash, the recoverer knows from what record in the log (known as the redo record) it should start the REDO operation, since any changes made to data files before that record are already on disk. After a checkpoint has been made, any log segments written before the undo records are no longer needed and can be recycled or removed. (When WAL-based BAR is implemented, the log segments would be archived before being recycled or removed.)

The checkpoint maker is also able to create a few log segments for future use, so as to avoid the need for `LogInsert` or `LogFlush` to spend time in creating them. (If that happens, the entire database system will be delayed by the creation operation, so it's better if the files can be created in the checkpoint maker, which is not on anyone's critical path.) By default a new 16MB segment file is created only if more than 75% of the current segment has been used. This is inadequate if the system generates more than 4MB of log output between checkpoints. One can instruct the server to pre-create up to 64 log segments at checkpoint time by modifying the `WAL_FILES` configuration parameter.

The postmaster spawns a special backend process every so often to create the next checkpoint. A checkpoint is created every `CHECKPOINT_SEGMENTS` log segments, or every `CHECKPOINT_TIMEOUT` seconds, whichever comes first. The default settings are 3 segments and 300 seconds respectively. It is also possible to force a checkpoint by using the SQL command `CHECKPOINT`.

Reducing `CHECKPOINT_SEGMENTS` and/or `CHECKPOINT_TIMEOUT` causes checkpoints to be done more often. This allows faster after-crash recovery (since less work will need to be redone). However, one must balance this against the increased cost of flushing dirty data pages more often. In addition, to ensure data page consistency, the first modification of a data page after each checkpoint results in logging the entire page content. Thus a smaller checkpoint interval increases the volume of output to the log, partially negating the goal of using a smaller interval, and in any case causing more disk I/O.

The number of 16MB segment files will always be at least `WAL_FILES` + 1, and will normally not exceed `WAL_FILES` + MAX(`WAL_FILES`, `CHECKPOINT_SEGMENTS`) + 1. This may be used to estimate space requirements for WAL. Ordinarily, when an old log segment files are no longer needed, they are recycled (renamed to become the next sequential future segments). If, due to a short-term peak of log output rate, there are more than `WAL_FILES` + MAX(`WAL_FILES`, `CHECKPOINT_SEGMENTS`) + 1 segment files, then unneeded segment files will be deleted instead of recycled until the system gets back under this limit. (If this happens on a regular basis, `WAL_FILES` should be increased to avoid it. Deleting log segments that will only have to be created again later is expensive and pointless.)

The COMMIT_DELAY parameter defines for how many microseconds the backend will sleep after writing a commit record to the log with LogInsert but before performing a LogFlush. This delay allows other backends to add their commit records to the log so as to have all of them flushed with a single log sync. No sleep will occur if fsync is not enabled or if fewer than COMMIT_SIBLINGS other backends are not currently in active transactions; this avoids sleeping when it's unlikely that any other backend will commit soon. Note that on most platforms, the resolution of a sleep request is ten milliseconds, so that any nonzero COMMIT_DELAY setting between 1 and 10000 microseconds will have the same effect. Good values for these parameters are not yet clear; experimentation is encouraged.

The WAL_SYNC_METHOD parameter determines how PostgreSQL will ask the kernel to force WAL updates out to disk. All the options should be the same as far as reliability goes, but it's quite platform-specific which one will be the fastest. Note that this parameter is irrelevant if FSYNC has been turned off.

Setting the WAL_DEBUG parameter to any nonzero value will result in each LogInsert and LogFlush WAL call being logged to standard error. At present, it makes no difference what the nonzero value is. This option may be replaced by a more general mechanism in the future.

Chapter 12. Disk Storage

This section needs to be written. Some information is in the FAQ. Volunteers? - thomas 1998-01-11

Chapter 13. Database Failures

Database failures (or the possibility of such) must be assumed to be lurking, ready to strike at some time in the future. A prudent database administrator will plan for the inevitability of failures of all possible kinds, and will have appropriate plans and procedures in place *before* the failure occurs.

Database recovery is necessary in the event of hardware or software failure. There are several categories of failures; some of these require relatively minor adjustments to the database, while others may depend on the existence of previously prepared database dumps and other recovery data sets. It should be emphasized that if your data is important and/or difficult to regenerate, then you should have considered and prepared for various failure scenarios.

13.1. Disk Filled

A filled data disk may result in subsequent corruption of database indexes, but not of the fundamental data tables. If the WAL files are on the same disk (as is the case for a default configuration) then a filled disk during database initialization may result in corrupted or incomplete WAL files. This failure condition is detected and the database will refuse to start up. You must free up additional space on the disk (or move the WAL area to another disk; see Section 11.3, “WAL Configuration”) and then restart the postmaster to recover from this condition.

13.2. Disk Failed

Failure of any disk (or of a logical storage device such as a RAID subsystem) involved with an active database will require that the database be recovered from a previously prepared database dump. This dump must be prepared using pg_dumpall, and updates to the database occurring after the database installation was dumped will be lost.

Chapter 14. Regression Tests

14.1. Introduction

The regression tests are a comprehensive set of tests for the SQL implementation in PostgreSQL. They test standard SQL operations as well as the extended capabilities of PostgreSQL. The test suite was originally developed by Jolly Chen and Andrew Yu, and was extensively revised and repackaged by Marc Fournier and Thomas Lockhart. From PostgreSQL 6.1 onward the regression tests are current for every official release.

14.2. Running the Tests

The regression test can be run against an already installed and running server, or using a temporary installation within the build tree. Furthermore, there is a "parallel" and a "sequential" mode for running the tests. The sequential method runs each test script in turn, whereas the parallel method starts up multiple server processes to run groups of tests in parallel. Parallel testing gives confidence that interprocess communication and locking are working correctly. For historical reasons, the sequential test is usually run against an existing installation and the parallel method against a temporary installation, but there are no technical reasons for this.

To run the regression tests after building but before installation, type

```
$ gmake check
```

in the top-level directory. (Or you can change to `src/test/regress` and run the command there.) This will first build several auxiliary files, such as platform-dependent "expected" files and some sample user-defined trigger functions, and then run the test driver script. At the end you should see something like

```
=======================
 All 77 tests passed.
=======================
```

or otherwise a note about what tests failed. See Section 14.3, "Test Evaluation" below for more.

Note

Because this test method runs a temporary server, it will not work when you are the root user (the server will not start as root). If you already did the build as root, you do not have to start all over. Instead, make the regression test directory writable by some other user, log in as that user, and restart the tests. For example,

```
root# chmod -R a+w src/test/regress
root# chmod -R a+w contrib/spi
root# su - joeuser
joeuser$ cd top-level build directory
joeuser$ gmake check
```

(The only possible "security risk" here is that other users might be able to alter the regression test results behind your back. Use common sense when managing user permissions.)

Alternatively, run the tests after installation.

Tip

The parallel regression test starts quite a few processes under your user ID. Presently, the maximum concurrency is twenty parallel test scripts, which means sixty processes --- there's a backend, a psql, and usually a shell parent process for the psql for each test script. So if your system enforces a per-user limit on the number of processes, make sure this limit is at least seventy-five or so, else you may get random-seeming failures in the parallel test. If you are not in a position to raise the limit, you can edit the file `src/test/regress/parallel_schedule` to split the larger concurrent test sets into more manageable groups.

Tip

On some systems, the default Bourne-compatible shell (`/bin/sh`) gets confused when it has to manage too many child processes in parallel. This may cause the parallel test run to lock up or fail. In such cases, specify a different Bourne-compatible shell on the command line, for example:

```
$ gmake SHELL=/bin/ksh check
```

If no non-broken shell is available, you can alter the parallel test schedule as suggested above.

To run the tests after installation (see Chapter 1, *Installation Instructions*), initialize a data area and start the server, as explained in Chapter 3, *Server Runtime Environment*, then type

```
$ gmake installcheck
```

The tests will expect to contact the server at the local host and the default port number, unless directed otherwise by `PGHOST` and `PGPORT` environment variables.

14.3. Test Evaluation

Some properly installed and fully functional PostgreSQL installations can "fail" some of these regression tests due to platform-specific artifacts such as varying floating-point representation and time zone support. The tests are currently evaluated using a simple diff comparison against the outputs generated on a reference system, so the results are sensitive to small system differences. When a test is reported as "failed", always examine the differences between expected and actual results; you may well find that the differences are not significant. Nonetheless, we still strive to maintain accurate reference files across all supported platforms, so it can be expected that all tests pass.

The actual outputs of the regression tests are in files in the `src/test/regress/results` directory. The test script uses diff to compare each output file against the reference outputs stored in the `src/test/regress/expected` directory. Any differences are saved for your inspection in `src/test/regress/regression.diffs`. (Or you can run diff yourself, if you prefer.)

14.3.1. Error message differences

Some of the regression tests involve intentional invalid input values. Error messages can come from either the PostgreSQL code or from the host platform system routines. In the latter case, the messages may vary between platforms, but should reflect similar information. These differences in messages will result in a "failed" regression test that can be validated by inspection.

14.3.2. Locale differences

The tests expect to run in plain "C" locale. This should not cause any problems when you run the tests against a temporary installation, since the regression test driver takes care to start the server in C locale.

However, if you run the tests against an already-installed server that is using non-C locale settings, you may see differences caused by varying rules for string sort order, formatting of numeric and monetary values, and so forth.

In some locales the resulting differences are small and easily checked by inspection. However, in a locale that changes the rules for formatting of numeric values (typically by swapping the usage of commas and decimal points), entry of some data values will fail, resulting in extensive differences later in the tests where the missing data values are supposed to be used.

14.3.3. Date and time differences

Some of the queries in the `timestamp` test will fail if you run the test on the day of a daylight-saving time changeover, or the day before or after one. These queries assume that the intervals between midnight yesterday, midnight today and midnight tomorrow are exactly twenty-four hours -- which is wrong if daylight-saving time went into or out of effect meanwhile.

Most of the date and time results are dependent on the time zone environment. The reference files are generated for time zone `PST8PDT` (Berkeley, California) and there will be apparent failures if the tests are not run with that time zone setting. The regression test driver sets environment variable `PGTZ` to `PST8PDT`, which normally ensures proper results. However, your system must provide library support for the `PST8PDT` time zone, or the time zone-dependent tests will fail. To verify that your machine does have this support, type the following:

```
$ env TZ=PST8PDT date
```

The command above should have returned the current system time in the `PST8PDT` time zone. If the `PST8PDT` database is not available, then your system may have returned the time in GMT. If the `PST8PDT` time zone is not available, you can set the time zone rules explicitly:

```
PGTZ='PST8PDT7,M04.01.0,M10.05.03'; export PGTZ
```

There appear to be some systems that do not accept the recommended syntax for explicitly setting the local time zone rules; you may need to use a different `PGTZ` setting on such machines.

Some systems using older time zone libraries fail to apply daylight-saving corrections to dates before 1970, causing pre-1970 PDT times to be displayed in PST instead. This will result in localized differences in the test results.

14.3.4. Floating-point differences

Some of the tests involve computing 64-bit (`double precision`) numbers from table columns. Differences in results involving mathematical functions of `double precision` columns have been observed. The float8 and geometry tests are particularly prone to small differences across platforms, or even with different compiler optimization options. Human eyeball comparison is needed to determine the real significance of these differences which are usually 10 places to the right of the decimal point.

Some systems signal errors from `pow()` and `exp()` differently from the mechanism expected by the current PostgreSQL code.

14.3.5. Polygon differences

Several of the tests involve operations on geographic data about the Oakland/Berkeley, California street map. The map data is expressed as polygons whose vertices are represented as pairs of `double precision` numbers (decimal latitude and longitude). Initially, some tables are created and loaded with

geographic data, then some views are created that join two tables using the polygon intersection operator (##), then a select is done on the view.

When comparing the results from different platforms, differences occur in the 2nd or 3rd place to the right of the decimal point. The SQL statements where these problems occur are the following:

```
SELECT * from street;
SELECT * from iexit;
```

14.3.6. Row ordering differences

You might see differences in which the same rows are output in a different order than what appears in the expected file. In most cases this is not, strictly speaking, a bug. Most of the regression test scripts are not so pedantic as to use an ORDER BY for every single SELECT, and so their result row orderings are not well-defined according to the letter of the SQL specification. In practice, since we are looking at the same queries being executed on the same data by the same software, we usually get the same result ordering on all platforms, and so the lack of ORDER BY isn't a problem. Some queries do exhibit cross-platform ordering differences, however. (Ordering differences can also be triggered by non-C locale settings.)

Therefore, if you see an ordering difference, it's not something to worry about, unless the query does have an ORDER BY that your result is violating. But please report it anyway, so that we can add an ORDER BY to that particular query and thereby eliminate the bogus “failure” in future releases.

You might wonder why we don't order all the regress test queries explicitly to get rid of this issue once and for all. The reason is that that would make the regression tests less useful, not more, since they'd tend to exercise query plan types that produce ordered results to the exclusion of those that don't.

14.3.7. The “random” test

There is at least one case in the “random” test script that is intended to produce random results. This causes random to fail the regression test once in a while (perhaps once in every five to ten trials). Typing

```
diff results/random.out expected/random.out
```

should produce only one or a few lines of differences. You need not worry unless the random test always fails in repeated attempts. (On the other hand, if the random test is *never* reported to fail even in many trials of the regression tests, you probably *should* worry.)

14.4. Platform-specific comparison files

Since some of the tests inherently produce platform-specific results, we have provided a way to supply platform-specific result comparison files. Frequently, the same variation applies to multiple platforms; rather than supplying a separate comparison file for every platform, there is a mapping file that defines which comparison file to use. So, to eliminate bogus test “failures” for a particular platform, you must choose or make a variant result file, and then add a line to the mapping file, which is `resultmap`.

Each line in the mapping file is of the form

```
testname/platformpattern=comparisonfilename
```

The test name is just the name of the particular regression test module. The platform pattern is a pattern in the style of expr (that is, a regular expression with an implicit ^ anchor at the start). It is matched against the platform name as printed by `config.guess` followed by `:gcc` or `:cc`, depending on whether

you use the GNU compiler or the system's native compiler (on systems where there is a difference). The comparison file name is the name of the substitute result comparison file.

For example: some systems using older time zone libraries fail to apply daylight-saving corrections to dates before 1970, causing pre-1970 PDT times to be displayed in PST instead. This causes a few differences in the `horology` regression test. Therefore, we provide a variant comparison file, `horology-no-DST-before-1970.out`, which includes the results to be expected on these systems. To silence the bogus "failure" message on HPPA platforms, `resultmap` includes

```
horology/hppa=horology-no-DST-before-1970
```

which will trigger on any machine for which the output of `config.guess` begins with "`hppa`". Other lines in `resultmap` select the variant comparison file for other platforms where it's appropriate.

Appendix A. Release Notes

A.1. Release 7.2.8

Release date

2005-05-09

This release contains a variety of fixes from 7.2.7, including one security-related issue.

A.1.1. Migration to version 7.2.8

A dump/restore is not required for those running 7.2.X.

A.1.2. Changes

- Repair ancient race condition that allowed a transaction to be seen as committed for some purposes (eg SELECT FOR UPDATE) slightly sooner than for other purposes

  This is an extremely serious bug since it could lead to apparent data inconsistencies being briefly visible to applications.

- Repair race condition between relation extension and VACUUM

  This could theoretically have caused loss of a page's worth of freshly-inserted data, although the scenario seems of very low probability. There are no known cases of it having caused more than an Assert failure.

- Fix `EXTRACT(EPOCH)` for `TIME WITH TIME ZONE` values

- Additional buffer overrun checks in plpgsql (Neil)

- Fix pg_dump to dump index names and trigger names containing `%` correctly (Neil)

- Prevent `to_char(interval)` from dumping core for month-related formats

- Fix `contrib/pgcrypto` for newer OpenSSL builds (Marko Kreen)

A.2. Release 7.2.7

Release date

2005-01-31

This release contains a variety of fixes from 7.2.6, including several security-related issues.

A.2.1. Migration to version 7.2.7

A dump/restore is not required for those running 7.2.X.

A.2.2. Changes

- Disallow `LOAD` to non-superusers

  On platforms that will automatically execute initialization functions of a shared library (this includes at least Windows and ELF-based Unixen), `LOAD` can be used to make the server execute arbitrary code. Thanks to NGS Software for reporting this.

- Add needed STRICT marking to some contrib functions (Kris Jurka)

- Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)

- Fix planning error for FULL and RIGHT outer joins

  The result of the join was mistakenly supposed to be sorted the same as the left input. This could not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright wrong answers.

- Fix display of negative intervals in SQL and GERMAN datestyles

A.3. Release 7.2.6

Release date

2004-10-22

This release contains a variety of fixes from 7.2.5.

A.3.1. Migration to version 7.2.6

A dump/restore is not required for those running 7.2.X.

A.3.2. Changes

- 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

A.4. Release 7.2.5

Release date

2004-08-16

This release contains a variety of fixes from 7.2.4.

A.4.1. Migration to version 7.2.5

A dump/restore is not required for those running 7.2.X.

A.4.2. Changes

- 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

A.5. 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.

A.5.1. Migration to version 7.2.4

A dump/restore is *not* required for those running version 7.2.*.

A.5.2. Changes

- 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.grolist 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()

A.6. 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.

A.6.1. Migration to version 7.2.3

A dump/restore is *not* required for those running version 7.2.*.

A.6.2. Changes

- 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 Niijima)
- Fix pg_dump to properly dump FULL JOIN USING (Tom)

A.7. Release 7.2.2

Release date

2002-08-23

This release contains a variety of fixes for version 7.2.1.

A.7.1. Migration to version 7.2.2

A dump/restore is *not* required for those running version 7.2.*.

A.7.2. Changes

- 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)

A.8. Release 7.2.1

Release date

2002-03-21

This release contains a variety of fixes for version 7.2.

A.8.1. Migration to version 7.2.1

A dump/restore is *not* required for those running version 7.2.

A.8.2. Changes

- 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)

A.9. Release 7.2

Release date

2002-02-04

A.9.1. Overview

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.

A.9.2. 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`.

A.9.3. Changes

A.9.3.1. 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)

A.9.3.2. 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)

A.9.3.3. Privileges

- 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)

A.9.3.4. 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)

A.9.3.5. 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)

A.9.3.6. Queries

- 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)

A.9.3.7. 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)

A.9.3.8. 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)

A.9.3.9. 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)

A.9.3.10. Internationalization

- 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)

A.9.3.11. 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)

A.9.3.12. PL/Perl

- New untrusted PL/Perl (Alex Pilosov)
- PL/Perl is now built on some platforms even if libperl is not shared (Peter E)

A.9.3.13. PL/Tcl

- Now reports errorInfo (Vsevolod Lobko)
- Add spi_lastoid function (bob@redivi.com)

A.9.3.14. PL/Python

- ...is new (Andrew Bosma)

A.9.3.15. 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)

A.9.3.16. libpq

- New function PQescapeString() to escape quotes in command strings (Florian Weimer)
- New function PQescapeBytea() escapes binary strings for use as SQL string literals

A.9.3.17. 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 isWriteable() 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)

A.9.3.18. 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 odbc.sql (Peter E)

A.9.3.19. 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)

A.9.3.20. 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)

A.9.3.21. 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)

A.9.3.22. 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 pgjindent 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)

A.9.3.23. 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)

A.10. Release 7.1.3

Release date

2001-08-15

A.10.1. Migration to version 7.1.3

A dump/restore is *not* required for those running 7.1.X.

A.10.2. Changes

```
Remove unused WAL segements 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)
```

A.11. Release 7.1.2

Release date

2001-05-11

This has one fix from 7.1.1.

A.11.1. Migration to version 7.1.2

A dump/restore is *not* required for those running 7.1.X.

A.11.2. Changes

```
Fix PL/pgSQL SELECTs when returning no rows
Fix for psql backslash core dump
Referential integrity privilege fix
Optimizer fixes
pg_dump cleanups
```

A.12. Release 7.1.1

Release date

2001-05-05

This has a variety of fixes from 7.1.

A.12.1. Migration to version 7.1.1

A dump/restore is *not* required for those running 7.1.

A.12.2. Changes

```
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)
```

A.13. 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.

A.13.1. Migration to version 7.1

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

A.13.2. Changes

```
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
```

A.14. Release 7.0.3

Release date

2000-11-11

This has a variety of fixes from 7.0.2.

A.14.1. Migration to version 7.0.3

A dump/restore is *not* required for those running 7.0.*.

A.14.2. Changes

```
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 aloocation 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)
```

A.15. Release 7.0.2

Release date

2000-06-05

This is a repackaging of 7.0.1 with added documentation.

A.15.1. Migration to version 7.0.2

A dump/restore is *not* required for those running 7.*.

A.15.2. Changes

```
Added documentation to tarball.
```

A.16. Release 7.0.1

Release date

2000-06-01

This is a cleanup release for 7.0.

A.16.1. Migration to version 7.0.1

A dump/restore is *not* required for those running 7.0.

A.16.2. Changes

```
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)
```

A.17. 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.

A.17.1. 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.

A.17.2. Changes

```
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 <mascarim@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 f1::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/reltime 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 > lowbound AND x <
 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 pgsql 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 lextest, 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
```

A.18. 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.

A.18.1. Migration to version 6.5.3

A dump/restore is *not* required for those running 6.5.*.

A.18.2. Changes

```
Updated version of pgaccess 0.98
NT-specific patch
Fix dumping rules on inherited tables
```

A.19. 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.

A.19.1. Migration to version 6.5.2

A dump/restore is *not* required for those running 6.5.*.

A.19.2. Changes

```
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 deparser(Tom)
Comment out FileUnlink of excess segments during mdtruncate()(Tom)
IRIX linking fix from Yu Cao >yucao@falcon.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
```

A.20. 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.

A.20.1. Migration to version 6.5.1

A dump/restore is *not* required for those running 6.5.

A.20.2. Changes

```
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 expresssion 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 ^ precidence(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)
```

A.21. 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.

A.21.1. 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.*

A.21.1.1. 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.

Note

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.

A.21.2. Changes

`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)
Ecpg fixes/features, see src/interfaces/ecpg/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 suppports 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 McMurry)
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)
```

A.22. Release 6.4.2

Release date

1998-12-20

The 6.4.1 release was improperly packaged. This also has one additional bug fix.

A.22.1. Migration to version 6.4.2

A dump/restore is *not* required for those running 6.4.*.

A.22.2. Changes

```
Fix for datetime constant problem on some platforms(Thomas)
```

A.23. 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.

A.23.1. Migration to version 6.4.1

A dump/restore is *not* required for those running 6.4.

A.23.2. Changes

```
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)
```

A.24. 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.

A.24.1. 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.

A.24.2. Changes

```
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 availble on the command line
Code for 64-bit integer supported added, configure tested, int8
 type(Thomas)
Prevent file descriptor leaf 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 supress quotes around indentifiers
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 POSTGRESDIR 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 documention(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)
```

A.25. 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.

A.25.1. Changes

```
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)
```

A.26. 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.

A.26.1. Changes

```
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 libptcl/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)
```

A.27. 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.

Note

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.

A.27.1. 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.

A.27.2. Changes

```
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 MergeJion(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, PGCOSTHEAP, PGCOSTINDEX, 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 istrue() 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)
```

A.28. 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.

A.28.1. 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 aggname = 'avg' and aggbasetype = 790;
```

This will need to be done to every existing database, including template1.

A.28.2. Changes

```
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)
```

A.29. Release 6.2

Release date

1997-10-02

A dump/restore is required for those wishing to migrate data from previous releases of PostgreSQL.

A.29.1. 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.

A.29.2. 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.

A.29.3. Changes

```
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/reltime 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)
```

A.30. Release 6.1.1

Release date

1997-07-22

A.30.1. 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.

A.30.2. Changes

```
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)
```

A.31. 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).

A.31.1. 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.

A.31.2. Changes

```
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 memmory leaks, cleanups (Vadim, Massimo)
fix storage manager memmory 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
```

A.32. Release 6.0

Release date

1997-01-29

A dump/restore is required for those wishing to migrate data from previous releases of PostgreSQL.

A.32.1. 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.

A.32.2. 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.

A.32.3. Changes

```
Bug Fixes
---------
ALTER TABLE bug - running postgress 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 unitialized 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 distrubution(Marc)
Idend 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 easly 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, centeralized #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
```

A.33. 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.

A.34. Release 1.02

Release date

1996-08-01

A.34.1. 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.

Note

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.

A.34.2. 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);
```

A.34.3. Changes

```
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 tck7.4/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
```

A.35. Release 1.01

Release date

1996-02-23

A.35.1. 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);
```

A.35.2. Changes

```
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
```

A.36. Release 1.0

Release date

1995-09-05

A.36.1. Changes

```
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
```

A.37. Postgres95 Release 0.03

Release date

1995-07-21

A.37.1. Changes

```
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.
```

A.38. Postgres95 Release 0.02

Release date

1995-05-25

A.38.1. Changes

```
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
```

A.39. Postgres95 Release 0.01

Release date

1995-05-01

Initial release.

PostgreSQL 7.2.8 Programmer's Guide

The PostgreSQL Global Development Group

PostgreSQL 7.2.8 Programmer's Guide

The PostgreSQL Global Development Group
Copyright © 1996-2001 The PostgreSQL Global Development Group

Legal Notice

PostgreSQL is Copyright © 1996-2001 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.

Table of Contents

I. Client Interfaces 1
1. libpq - C Library 5
1.1. Introduction 5
1.2. Database Connection Functions 5
1.3. Command Execution Functions 11
1.4. Asynchronous Query Processing 16
1.5. The Fast-Path Interface 19
1.6. Asynchronous Notification 19
1.7. Functions Associated with the COPY Command 20
1.8. libpq Tracing Functions 22
1.9. libpq Control Functions 22
1.10. Environment Variables 23
1.11. Threading Behavior 24
1.12. Building Libpq Programs 24
1.13. Example Programs 25
2. Large Objects 35
2.1. Introduction 35
2.2. Implementation Features 35
2.3. Interfaces 35
2.4. Server-side Built-in Functions 37
2.5. Accessing Large Objects from Libpq 38
3. libpq++ - C++ Binding Library 43
3.1. Introduction 43
3.2. Control and Initialization 43
3.3. libpq++ Classes 44
3.4. Database Connection Functions 44
3.5. Query Execution Functions 45
3.6. Asynchronous Notification 48
3.7. Functions Associated with the COPY Command 49
4. pgtcl - Tcl Binding Library 50
4.1. Introduction 50
4.2. Loading pgtcl into your application 51
4.3. pgtcl Command Reference Information 51
5. libpgeasy - Simplified C Library 71
6. ecpg - Embedded SQL in C 72
6.1. Why Embedded SQL? 72
6.2. The Concept 72
6.3. How To Use ecpg 72
6.4. Limitations 76
6.5. Porting From Other RDBMS Packages 76
6.6. For the Developer 76
7. ODBC Interface 81
7.1. Introduction 81
7.2. Installation 81
7.3. Configuration Files 82
7.4. Windows Applications 83
7.5. ApplixWare 84
8. JDBC Interface 88
8.1. Setting up the JDBC Driver 88
8.2. Using the Driver 89
8.3. Issuing a Query and Processing the Result 91

8.4. Performing Updates 92
8.5. Creating and Modifying Database Objects 93
8.6. Storing Binary Data 93
8.7. PostgreSQL Extensions to the JDBC API 96
8.8. Using the driver in a multi-threaded or a servlet environment 116
8.9. Further Reading 117
9. PyGreSQL - Python Interface 118
9.1. The pg Module 118
9.2. pg Module Functions 119
9.3. Connection object: pgobject 131
9.4. Database wrapper class: DB 144
9.5. Query result object: pgqueryobject 154
9.6. Large Object: pglarge 160
9.7. DB-API Interface 170
II. Server Programming 171
10. Architecture 174
10.1. PostgreSQL Architectural Concepts 174
11. Extending SQL: An Overview 177
11.1. How Extensibility Works 177
11.2. The PostgreSQL Type System 177
11.3. About the PostgreSQL System Catalogs 177
12. Extending SQL: Functions 181
12.1. Introduction 181
12.2. Query Language (SQL) Functions 181
12.3. Procedural Language Functions 185
12.4. Internal Functions 185
12.5. C Language Functions 186
12.6. Function Overloading 198
12.7. Procedural Language Handlers 199
13. Extending SQL: Types 202
14. Extending SQL: Operators 204
14.1. Introduction 204
14.2. Example 204
14.3. Operator Optimization Information 205
15. Extending SQL: Aggregates 209
16. The Rule System 211
16.1. Introduction 211
16.2. What is a Query Tree? 211
16.3. Views and the Rule System 213
16.4. Rules on INSERT, UPDATE and DELETE 220
16.5. Rules and Permissions 230
16.6. Rules versus Triggers 231
17. Interfacing Extensions To Indexes 234
17.1. Introduction 234
17.2. Access Methods 234
17.3. Access Method Strategies 235
17.4. Access Method Support Routines 235
17.5. Operator Classes 236
17.6. Creating the Operators and Support Routines 237
18. Index Cost Estimation Functions 240
19. GiST Indexes 243
20. Triggers 245
20.1. Trigger Creation 245
20.2. Interaction with the Trigger Manager 246

20.3. Visibility of Data Changes 248
20.4. Examples 249
21. Server Programming Interface 252
21.1. Interface Functions 252
21.2. Interface Support Functions 266
21.3. Memory Management 273
21.4. Visibility of Data Changes 285
21.5. Examples 285
III. Procedural Languages 289
22. Procedural Languages 292
22.1. Introduction 292
22.2. Installing Procedural Languages 292
23. PL/pgSQL - SQL Procedural Language 294
23.1. Overview 294
23.2. Structure of PL/pgSQL 296
23.3. Declarations 297
23.4. Expressions 300
23.5. Basic Statements 301
23.6. Control Structures 304
23.7. Cursors 309
23.8. Errors and Messages 311
23.9. Trigger Procedures 312
23.10. Examples 314
23.11. Porting from Oracle PL/SQL 315
24. PL/Tcl - Tcl Procedural Language 327
24.1. Overview 327
24.2. Description 327
25. PL/Perl - Perl Procedural Language 334
25.1. Overview 334
25.2. Building and Installing PL/Perl 334
25.3. Description 335
26. PL/Python - Python Procedural Language 338
26.1. Introduction 338
26.2. Installation 338
26.3. Using PL/Python 338

List of Figures

10.1. How a connection is established 175
11.1. The major PostgreSQL system catalogs 179

List of Tables

4.1. `pgtcl` Commands 50
11.1. Catalogs 178
12.1. Equivalent C Types 187
17.1. Index Access Method Schema 234
17.2. B-tree 235
23.1. Single Quotes Escaping Chart 315

List of Examples

1.1. libpq Example Program 1 25
1.2. libpq Example Program 2 28
1.3. libpq Example Program 3 30
2.1. Large Objects with Libpq Example Program 38
4.1. pgtcl Example Program 50
8.1. Processing a Simple Query in JDCB 91
8.2. Simple Delete Example 92
8.3. Drop Table Example 93
8.4. Binary Data Examples 94
22.1. Manual Installation of PL/pgSQL 293
23.1. A PL/pgSQL Trigger Procedure Example 313
23.2. A Simple PL/pgSQL Function to Increment an Integer 314
23.3. A Simple PL/pgSQL Function to Concatenate Text 314
23.4. A PL/pgSQL Function on Composite Type 314
23.5. A Simple Function 316
23.6. A Function that Creates Another Function 317
23.7. A Procedure with a lot of String Manipulation and OUT Parameters 318

Part I. Client Interfaces

This part of the manual is the description of the client-side programming interfaces and support libraries for various languages.

Table of Contents

1. libpq - C Library 5
    1.1. Introduction 5
    1.2. Database Connection Functions 5
    1.3. Command Execution Functions 11
        1.3.1. Main Routines 11
        1.3.2. Escaping strings for inclusion in SQL queries 13
        1.3.3. Escaping binary strings for inclusion in SQL queries 13
        1.3.4. Retrieving SELECT Result Information 14
        1.3.5. Retrieving SELECT Result Values 15
        1.3.6. Retrieving Non-SELECT Result Information 16
    1.4. Asynchronous Query Processing 16
    1.5. The Fast-Path Interface 19
    1.6. Asynchronous Notification 19
    1.7. Functions Associated with the COPY Command 20
    1.8. libpq Tracing Functions 22
    1.9. libpq Control Functions 22
    1.10. Environment Variables 23
    1.11. Threading Behavior 24
    1.12. Building Libpq Programs 24
    1.13. Example Programs 25
2. Large Objects 35
    2.1. Introduction 35
    2.2. Implementation Features 35
    2.3. Interfaces 35
        2.3.1. Creating a Large Object 36
        2.3.2. Importing a Large Object 36
        2.3.3. Exporting a Large Object 36
        2.3.4. Opening an Existing Large Object 36
        2.3.5. Writing Data to a Large Object 36
        2.3.6. Reading Data from a Large Object 37
        2.3.7. Seeking on a Large Object 37
        2.3.8. Closing a Large Object Descriptor 37
        2.3.9. Removing a Large Object 37
    2.4. Server-side Built-in Functions 37
    2.5. Accessing Large Objects from Libpq 38
3. libpq++ - C++ Binding Library 43
    3.1. Introduction 43
    3.2. Control and Initialization 43
        3.2.1. Environment Variables 43
    3.3. libpq++ Classes 44
        3.3.1. Connection Class: `PgConnection` 44
        3.3.2. Database Class: `PgDatabase` 44
    3.4. Database Connection Functions 44
    3.5. Query Execution Functions 45
        3.5.1. Main Routines 45
        3.5.2. Retrieving SELECT Result Information 46
        3.5.3. Retrieving SELECT Result Values 47
        3.5.4. Retrieving Non-SELECT Result Information 48
    3.6. Asynchronous Notification 48
    3.7. Functions Associated with the COPY Command 49
4. pgtcl - Tcl Binding Library 50

4.1. Introduction 50
4.2. Loading pgtcl into your application 51
4.3. pgtcl Command Reference Information 51
5. libpgeasy - Simplified C Library 71
6. ecpg - Embedded SQL in C 72
6.1. Why Embedded SQL? 72
6.2. The Concept 72
6.3. How To Use ecpg 72
6.3.1. Preprocessor 72
6.3.2. Library 72
6.3.3. Error handling 73
6.4. Limitations 76
6.5. Porting From Other RDBMS Packages 76
6.6. For the Developer 76
6.6.1. The Preprocessor 76
6.6.2. A Complete Example 79
6.6.3. The Library 80
7. ODBC Interface 81
7.1. Introduction 81
7.2. Installation 81
7.3. Configuration Files 82
7.4. Windows Applications 83
7.4.1. Writing Applications 83
7.5. ApplixWare 84
7.5.1. Configuration 84
7.5.2. Common Problems 85
7.5.3. Debugging ApplixWare ODBC Connections 85
7.5.4. Running the ApplixWare Demo 86
7.5.5. Useful Macros 87
8. JDBC Interface 88
8.1. Setting up the JDBC Driver 88
8.1.1. Getting the Driver 88
8.1.2. Setting up the Class Path 89
8.1.3. Preparing the Database for JDBC 89
8.2. Using the Driver 89
8.2.1. Importing JDBC 89
8.2.2. Loading the Driver 90
8.2.3. Connecting to the Database 90
8.2.4. Closing the Connection 91
8.3. Issuing a Query and Processing the Result 91
8.3.1. Using the `Statement` or `PreparedStatement` Interface 92
8.3.2. Using the `ResultSet` Interface 92
8.4. Performing Updates 92
8.5. Creating and Modifying Database Objects 93
8.6. Storing Binary Data 93
8.7. PostgreSQL Extensions to the JDBC API 96
8.7.1. Accessing the Extensions 96
8.7.2. Geometric Data Types 100
8.7.3. Large Objects 113
8.8. Using the driver in a multi-threaded or a servlet environment 116
8.9. Further Reading 117
9. PyGreSQL - Python Interface 118
9.1. The `pg` Module 118
9.1.1. Constants 118

9.2. pg Module Functions 119
9.3. Connection object: pgobject 131
9.4. Database wrapper class: DB 144
9.5. Query result object: pgqueryobject 154
9.6. Large Object: pglarge 160
9.7. DB-API Interface 170

Chapter 1. libpq - C Library

1.1. Introduction

libpq is the C application programmer's interface to PostgreSQL. libpq is a set of library routines that allow client programs to pass queries to the PostgreSQL backend server and to receive the results of these queries. libpq is also the underlying engine for several other PostgreSQL application interfaces, including libpq++ (C++), `libpgtcl` (Tcl), Perl, and `ecpg`. So some aspects of libpq's behavior will be important to you if you use one of those packages.

Three short programs are included at the end of this section to show how to write programs that use `libpq`. There are several complete examples of `libpq` applications in the following directories:

```
src/test/examples
src/bin/psql
```

Frontend programs that use `libpq` must include the header file `libpq-fe.h` and must link with the `libpq` library.

1.2. Database Connection Functions

The following routines deal with making a connection to a PostgreSQL backend server. The application program can have several backend connections open at one time. (One reason to do that is to access more than one database.) Each connection is represented by a `PGconn` object which is obtained from `PQconnectdb` or `PQsetdbLogin`. Note that these functions will always return a non-null object pointer, unless perhaps there is too little memory even to allocate the `PGconn` object. The `PQstatus` function should be called to check whether a connection was successfully made before queries are sent via the connection object.

- `PQconnectdb` Makes a new connection to the database server.

  ```
  PGconn *PQconnectdb(const char *conninfo)
  ```

  This routine opens a new database connection using the parameters taken from the string `conninfo`. Unlike `PQsetdbLogin` below, the parameter set can be extended without changing the function signature, so use either of this routine or the nonblocking analogues `PQconnectStart` and `PQconnectPoll` is preferred for application programming. The passed string can be empty to use all default parameters, or it can contain one or more parameter settings separated by whitespace.

  Each parameter setting is in the form `keyword = value`. (To write an empty value or a value containing spaces, surround it with single quotes, e.g., `keyword = 'a value'`. Single quotes and backslashes within the value must be escaped with a backslash, e.g., `\'` or `\\`.) Spaces around the equal sign are optional. The currently recognized parameter keywords are:

  `host`

  Name of host to connect to. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is the name of the directory in which the socket file is stored. The default is to connect to a Unix-domain socket in `/tmp`.

`hostaddr`

IP address of host to connect to. This should be in standard numbers-and-dots form, as used by the BSD functions `inet_aton` et al. If a nonzero-length string is specified, TCP/IP communication is used.

Using `hostaddr` instead of host allows the application to avoid a host name look-up, which may be important in applications with time constraints. However, Kerberos authentication requires the host name. The following therefore applies. If host is specified without `hostaddr`, a host name lookup is forced. If `hostaddr` is specified without host, the value for `hostaddr` gives the remote address; if Kerberos is used, this causes a reverse name query. If both host and `hostaddr` are specified, the value for `hostaddr` gives the remote address; the value for host is ignored, unless Kerberos is used, in which case that value is used for Kerberos authentication. Note that authentication is likely to fail if libpq is passed a host name that is not the name of the machine at `hostaddr`.

Without either a host name or host address, libpq will connect using a local Unix domain socket.

`port`

Port number to connect to at the server host, or socket file name extension for Unix-domain connections.

`dbname`

The database name.

`user`

User name to connect as.

`password`

Password to be used if the server demands password authentication.

`options`

Trace/debug options to be sent to the server.

`tty`

A file or tty for optional debug output from the backend.

`requiressl`

Set to 1 to require SSL connection to the backend. Libpq will then refuse to connect if the server does not support SSL. Set to 0 (default) to negotiate with server.

If any parameter is unspecified, then the corresponding environment variable (see Section 1.10, “Environment Variables”) is checked. If the environment variable is not set either, then hardwired defaults are used. The return value is a pointer to an abstract struct representing the connection to the backend.

- `PQsetdbLogin` Makes a new connection to the database server.

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
```

```
                        const char *pgoptions,
                        const char *pgtty,
                        const char *dbName,
                        const char *login,
                        const char *pwd)
```

This is the predecessor of `PQconnectdb` with a fixed number of parameters but the same functionality.

- `PQsetdb` Makes a new connection to the database server.

  ```
  PGconn *PQsetdb(char *pghost,
                  char *pgport,
                  char *pgoptions,
                  char *pgtty,
                  char *dbName)
  ```

  This is a macro that calls `PQsetdbLogin` with null pointers for the *`login`* and *`pwd`* parameters. It is provided primarily for backward compatibility with old programs.

- `PQconnectStart`, `PQconnectPoll` Make a connection to the database server in a nonblocking manner.

  ```
  PGconn *PQconnectStart(const char *conninfo)

  PostgresPollingStatusType PQconnectPoll(PGconn *conn)
  ```

  These two routines are used to open a connection to a database server such that your application's thread of execution is not blocked on remote I/O whilst doing so.

  The database connection is made using the parameters taken from the string `conninfo`, passed to `PQconnectStart`. This string is in the same format as described above for `PQconnectdb`.

  Neither `PQconnectStart` nor `PQconnectPoll` will block, as long as a number of restrictions are met:

  - The `hostaddr` and `host` parameters are used appropriately to ensure that name and reverse name queries are not made. See the documentation of these parameters under `PQconnectdb` above for details.

  - If you call `PQtrace`, ensure that the stream object into which you trace will not block.

  - You ensure for yourself that the socket is in the appropriate state before calling `PQconnectPoll`, as described below.

  To begin, call `conn=PQconnectStart("`*`connection_info_string`*`")`. If `conn` is NULL, then libpq has been unable to allocate a new `PGconn` structure. Otherwise, a valid `PGconn` pointer is returned (though not yet representing a valid connection to the database). On return from `PQconnectStart`, call `status=PQstatus(conn)`. If status equals `CONNECTION_BAD`, `PQconnectStart` has failed.

  If `PQconnectStart` succeeds, the next stage is to poll libpq so that it may proceed with the connection sequence. Loop thus: Consider a connection “inactive” by default. If `PQconnectPoll` last returned `PGRES_POLLING_ACTIVE`, consider it “active” instead. If `PQconnectPoll(conn)` last returned `PGRES_POLLING_READING`, perform a `select()` for reading on `PQsocket(conn)`. If it last returned `PGRES_POLLING_WRITING`, perform a `select()` for writing on `PQsocket(conn)`. If you have yet to call `PQconnectPoll`, i.e. after the call to `PQconnectStart`, behave as if it last

returned PGRES_POLLING_WRITING. If the select() shows that the socket is ready, consider it "active". If it has been decided that this connection is "active", call PQconnectPoll(conn) again. If this call returns PGRES_POLLING_FAILED, the connection procedure has failed. If this call returns PGRES_POLLING_OK, the connection has been successfully made.

Note that the use of select() to ensure that the socket is ready is merely a (likely) example; those with other facilities available, such as a poll() call, may of course use that instead.

At any time during connection, the status of the connection may be checked, by calling PQstatus. If this is CONNECTION_BAD, then the connection procedure has failed; if this is CONNECTION_OK, then the connection is ready. Either of these states should be equally detectable from the return value of PQconnectPoll, as above. Other states may be shown during (and only during) an asynchronous connection procedure. These indicate the current stage of the connection procedure, and may be useful to provide feedback to the user for example. These statuses may include:

CONNECTION_STARTED

Waiting for connection to be made.

CONNECTION_MADE

Connection OK; waiting to send.

CONNECTION_AWAITING_RESPONSE

Waiting for a response from the server.

CONNECTION_AUTH_OK

Received authentication; waiting for connection start-up to continue.

CONNECTION_SETENV

Negotiating environment (part of the connection start-up).

Note that, although these constants will remain (in order to maintain compatibility), an application should never rely upon these appearing in a particular order, or at all, or on the status always being one of these documented values. An application may do something like this:

```
switch(PQstatus(conn))
{
    case CONNECTION_STARTED:
        feedback = "Connecting...";
        break;

    case CONNECTION_MADE:
        feedback = "Connected to server...";
        break;
.
.
.
    default:
        feedback = "Connecting...";
}
```

Note that if PQconnectStart returns a non-NULL pointer, you must call PQfinish when you are finished with it, in order to dispose of the structure and any associated memory blocks. This must be done even if a call to PQconnectStart or PQconnectPoll failed.

PQconnectPoll will currently block if libpq is compiled with USE_SSL defined. This restriction may be removed in the future.

These functions leave the socket in a nonblocking state as if PQsetnonblocking had been called.

- PQconndefaults Returns the default connection options.

```
PQconninfoOption *PQconndefaults(void)

struct PQconninfoOption
{
    char   *keyword;   /* The keyword of the option */
    char   *envvar;    /* Fallback environment variable name */
    char   *compiled;  /* Fallback compiled in default value */
    char   *val;       /* Option's current value, or NULL */
    char   *label;     /* Label for field in connect dialog */
    char   *dispchar;  /* Character to display for this field
                          in a connect dialog. Values are:
                          ""        Display entered value as is
                          "*"       Password field - hide value
                          "D"       Debug option - don't show by
 default */
    int     dispsize;  /* Field size in characters for dialog */
}
```

Returns a connection options array. This may be used to determine all possible PQconnectdb options and their current default values. The return value points to an array of PQconninfoOption structs, which ends with an entry having a NULL keyword pointer. Note that the default values (val fields) will depend on environment variables and other context. Callers must treat the connection options data as read-only.

After processing the options array, free it by passing it to PQconninfoFree. If this is not done, a small amount of memory is leaked for each call to PQconndefaults.

In PostgreSQL versions before 7.0, PQconndefaults returned a pointer to a static array, rather than a dynamically allocated array. That was not thread-safe, so the behavior has been changed.

- PQfinish Close the connection to the backend. Also frees memory used by the PGconn object.

```
void PQfinish(PGconn *conn)
```

Note that even if the backend connection attempt fails (as indicated by PQstatus), the application should call PQfinish to free the memory used by the PGconn object. The PGconn pointer should not be used after PQfinish has been called.

- PQreset Reset the communication port with the backend.

```
void PQreset(PGconn *conn)
```

This function will close the connection to the backend and attempt to reestablish a new connection to the same server, using all the same parameters previously used. This may be useful for error recovery if a working connection is lost.

- `PQresetStart` `PQresetPoll` Reset the communication port with the backend, in a nonblocking manner.

  ```
  int PQresetStart(PGconn *conn);

  PostgresPollingStatusType PQresetPoll(PGconn *conn);
  ```

  These functions will close the connection to the backend and attempt to reestablish a new connection to the same server, using all the same parameters previously used. This may be useful for error recovery if a working connection is lost. They differ from `PQreset` (above) in that they act in a nonblocking manner. These functions suffer from the same restrictions as `PQconnectStart` and `PQconnectPoll`.

  Call `PQresetStart`. If it returns 0, the reset has failed. If it returns 1, poll the reset using `PQresetPoll` in exactly the same way as you would create the connection using `PQconnectPoll`.

libpq application programmers should be careful to maintain the `PGconn` abstraction. Use the accessor functions below to get at the contents of `PGconn`. Avoid directly referencing the fields of the `PGconn` structure because they are subject to change in the future. (Beginning in PostgreSQL release 6.4, the definition of struct `PGconn` is not even provided in `libpq-fe.h`. If you have old code that accesses `PGconn` fields directly, you can keep using it by including `libpq-int.h` too, but you are encouraged to fix the code soon.)

- `PQdb` Returns the database name of the connection.

  ```
  char *PQdb(const PGconn *conn)
  ```

  `PQdb` and the next several functions return the values established at connection. These values are fixed for the life of the `PGconn` object.

- `PQuser` Returns the user name of the connection.

  ```
  char *PQuser(const PGconn *conn)
  ```

- `PQpass` Returns the password of the connection.

  ```
  char *PQpass(const PGconn *conn)
  ```

- `PQhost` Returns the server host name of the connection.

  ```
  char *PQhost(const PGconn *conn)
  ```

- `PQport` Returns the port of the connection.

  ```
  char *PQport(const PGconn *conn)
  ```

- `PQtty` Returns the debug tty of the connection.

  ```
  char *PQtty(const PGconn *conn)
  ```

- `PQoptions` Returns the backend options used in the connection.

  ```
  char *PQoptions(const PGconn *conn)
  ```

- `PQstatus` Returns the status of the connection.

```
ConnStatusType PQstatus(const PGconn *conn)
```

The status can be one of a number of values. However, only two of these are seen outside of an asynchronous connection procedure - `CONNECTION_OK` or `CONNECTION_BAD`. A good connection to the database has the status `CONNECTION_OK`. A failed connection attempt is signaled by status `CONNECTION_BAD`. Ordinarily, an OK status will remain so until `PQfinish`, but a communications failure might result in the status changing to `CONNECTION_BAD` prematurely. In that case the application could try to recover by calling `PQreset`.

See the entry for `PQconnectStart` and `PQconnectPoll` with regards to other status codes that might be seen.

- `PQerrorMessage` Returns the error message most recently generated by an operation on the connection.

```
char *PQerrorMessage(const PGconn* conn);
```

Nearly all libpq functions will set `PQerrorMessage` if they fail. Note that by libpq convention, a non-empty `PQerrorMessage` will include a trailing newline.

- `PQbackendPID` Returns the process ID of the backend server handling this connection.

```
int PQbackendPID(const PGconn *conn);
```

The backend PID is useful for debugging purposes and for comparison to NOTIFY messages (which include the PID of the notifying backend). Note that the PID belongs to a process executing on the database server host, not the local host!

- `PQgetssl` Returns the SSL structure used in the connection, or NULL if SSL is not in use.

```
SSL *PQgetssl(const PGconn *conn);
```

This structure can be used to verify encryption levels, check server certificate and more. Refer to the SSL documentation for information about this structure.

You must define `USE_SSL` in order to get the prototype for this function. Doing this will also automatically include `ssl.h` from OpenSSL.

1.3. Command Execution Functions

Once a connection to a database server has been successfully established, the functions described here are used to perform SQL queries and commands.

1.3.1. Main Routines

- `PQexec` Submit a command to the server and wait for the result.

```
PGresult *PQexec(PGconn *conn,
                 const char *query);
```

Returns a `PGresult` pointer or possibly a NULL pointer. A non-NULL pointer will generally be returned except in out-of-memory conditions or serious errors such as inability to send the command

to the backend. If a NULL is returned, it should be treated like a PGRES_FATAL_ERROR result. Use PQerrorMessage to get more information about the error.

The PGresult structure encapsulates the result returned by the backend. libpq application programmers should be careful to maintain the PGresult abstraction. Use the accessor functions below to get at the contents of PGresult. Avoid directly referencing the fields of the PGresult structure because they are subject to change in the future. (Beginning in PostgreSQL 6.4, the definition of struct PGresult is not even provided in libpq-fe.h. If you have old code that accesses PGresult fields directly, you can keep using it by including libpq-int.h too, but you are encouraged to fix the code soon.)

- PQresultStatus Returns the result status of the command.

  ```
  ExecStatusType PQresultStatus(const PGresult *res)
  ```

  PQresultStatus can return one of the following values:

  - PGRES_EMPTY_QUERY -- The string sent to the backend was empty.
  - PGRES_COMMAND_OK -- Successful completion of a command returning no data
  - PGRES_TUPLES_OK -- The query successfully executed
  - PGRES_COPY_OUT -- Copy Out (from server) data transfer started
  - PGRES_COPY_IN -- Copy In (to server) data transfer started
  - PGRES_BAD_RESPONSE -- The server's response was not understood
  - PGRES_NONFATAL_ERROR
  - PGRES_FATAL_ERROR

  If the result status is PGRES_TUPLES_OK, then the routines described below can be used to retrieve the rows returned by the query. Note that a SELECT command that happens to retrieve zero rows still shows PGRES_TUPLES_OK. PGRES_COMMAND_OK is for commands that can never return rows (INSERT, UPDATE, etc.). A response of PGRES_EMPTY_QUERY often exposes a bug in the client software.

- PQresStatus Converts the enumerated type returned by PQresultStatus into a string constant describing the status code.

  ```
  char *PQresStatus(ExecStatusType status);
  ```

- PQresultErrorMessage returns the error message associated with the query, or an empty string if there was no error.

  ```
  char *PQresultErrorMessage(const PGresult *res);
  ```

  Immediately following a PQexec or PQgetResult call, PQerrorMessage (on the connection) will return the same string as PQresultErrorMessage (on the result). However, a PGresult will retain its error message until destroyed, whereas the connection's error message will change when subsequent operations are done. Use PQresultErrorMessage when you want to know the status associated with a particular PGresult; use PQerrorMessage when you want to know the status from the latest operation on the connection.

- PQclear Frees the storage associated with the PGresult. Every query result should be freed via PQclear when it is no longer needed.

  ```
  void PQclear(PQresult *res);
  ```

You can keep a PGresult object around for as long as you need it; it does not go away when you issue a new query, nor even if you close the connection. To get rid of it, you must call PQclear. Failure to do this will result in memory leaks in the frontend application.

- PQmakeEmptyPGresult Constructs an empty PGresult object with the given status.

  ```
  PGresult* PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
  ```

  This is libpq's internal routine to allocate and initialize an empty PGresult object. It is exported because some applications find it useful to generate result objects (particularly objects with error status) themselves. If *conn* is not NULL and status indicates an error, the connection's current errorMessage is copied into the PGresult. Note that PQclear should eventually be called on the object, just as with a PGresult returned by libpq itself.

1.3.2. Escaping strings for inclusion in SQL queries

PQescapeString Escapes a string for use within an SQL query.

```
size_t PQescapeString (char *to, const char *from, size_t length);
```

If you want to include strings that have been received from a source that is not trustworthy (for example, because a random user entered them), you cannot directly include them in SQL queries for security reasons. Instead, you have to quote special characters that are otherwise interpreted by the SQL parser.

PQescapeString performs this operation. The *from* points to the first character of the string that is to be escaped, and the *length* parameter counts the number of characters in this string (a terminating zero byte is neither necessary nor counted). *to* shall point to a buffer that is able to hold at least one more character than twice the value of *length*, otherwise the behavior is undefined. A call to PQescapeString writes an escaped version of the *from* string to the *to* buffer, replacing special characters so that they cannot cause any harm, and adding a terminating zero byte. The single quotes that must surround PostgreSQL string literals are not part of the result string.

PQescapeString returns the number of characters written to *to*, not including the terminating zero byte. Behavior is undefined when the *to* and *from* strings overlap.

1.3.3. Escaping binary strings for inclusion in SQL queries

PQescapeBytea Escapes a binary string (bytea type) for use within an SQL query.

```
unsigned char *PQescapeBytea(unsigned char *from,
                                        size_t from_length,
                                        size_t *to_length);
```

Certain ASCII characters *must* be escaped (but all characters *may* be escaped) when used as part of a bytea string literal in an SQL statement. In general, to escape a character, it is converted into the three digit octal number equal to the decimal ASCII value, and preceded by two backslashes. The single quote (') and backslash (\) characters have special alternate escape sequences. See the *User's Guide* for more information. PQescapeBytea performs this operation, escaping only the minimally required characters.

The *from* parameter points to the first character of the string that is to be escaped, and the *from_length* parameter reflects the number of characters in this binary string (a terminating zero byte is neither necessary

nor counted). The *to_length* parameter shall point to a buffer suitable to hold the resultant escaped string length. The result string length does not include the terminating zero byte of the result.

PQescapeBytea returns an escaped version of the *from* parameter binary string, to a caller-provided buffer. The return string has all special characters replaced so that they can be properly processed by the PostgreSQL string literal parser, and the bytea input function. A terminating zero byte is also added. The single quotes that must surround PostgreSQL string literals are not part of the result string.

1.3.4. Retrieving SELECT Result Information

- PQntuples Returns the number of tuples (rows) in the query result.

  ```
  int PQntuples(const PGresult *res);
  ```

- PQnfields Returns the number of fields (columns) in each row of the query result.

  ```
  int PQnfields(const PGresult *res);
  ```

- PQfname Returns the field (column) name associated with the given field index. Field indices start at 0.

  ```
  char *PQfname(const PGresult *res,
                int field_index);
  ```

- PQfnumber Returns the field (column) index associated with the given field name.

  ```
  int PQfnumber(const PGresult *res,
                const char *field_name);
  ```

  -1 is returned if the given name does not match any field.

- PQftype Returns the field type associated with the given field index. The integer returned is an internal coding of the type. Field indices start at 0.

  ```
  Oid PQftype(const PGresult *res,
              int field_index);
  ```

  You can query the system table pg_type to obtain the name and properties of the various data types. The OIDs of the built-in data types are defined in src/include/catalog/pg_type.h in the source tree.

- PQfmod Returns the type-specific modification data of the field associated with the given field index. Field indices start at 0.

  ```
  int PQfmod(const PGresult *res,
             int field_index);
  ```

- PQfsize Returns the size in bytes of the field associated with the given field index. Field indices start at 0.

  ```
  int PQfsize(const PGresult *res,
              int field_index);
  ```

  PQfsize returns the space allocated for this field in a database tuple, in other words the size of the server's binary representation of the data type. -1 is returned if the field is variable size.

- PQbinaryTuples Returns 1 if the PGresult contains binary tuple data, 0 if it contains ASCII data.

  ```
  int PQbinaryTuples(const PGresult *res);
  ```

Currently, binary tuple data can only be returned by a query that extracts data from a binary cursor.

1.3.5. Retrieving SELECT Result Values

- `PQgetvalue` Returns a single field (column) value of one tuple (row) of a `PGresult`. Tuple and field indices start at 0.

```
char* PQgetvalue(const PGresult *res,
                 int tup_num,
                 int field_num);
```

For most queries, the value returned by `PQgetvalue` is a null-terminated character string representation of the attribute value. But if `PQbinaryTuples()` is 1, the value returned by `PQgetvalue` is the binary representation of the type in the internal format of the backend server (but not including the size word, if the field is variable-length). It is then the programmer's responsibility to cast and convert the data to the correct C type. The pointer returned by `PQgetvalue` points to storage that is part of the `PGresult` structure. One should not modify it, and one must explicitly copy the value into other storage if it is to be used past the lifetime of the `PGresult` structure itself.

- `PQgetisnull` Tests a field for a NULL entry. Tuple and field indices start at 0.

```
int PQgetisnull(const PGresult *res,
                int tup_num,
                int field_num);
```

This function returns 1 if the field contains a NULL, 0 if it contains a non-null value. (Note that `PQgetvalue` will return an empty string, not a null pointer, for a NULL field.)

- `PQgetlength` Returns the length of a field (attribute) value in bytes. Tuple and field indices start at 0.

```
int PQgetlength(const PGresult *res,
                int tup_num,
                int field_num);
```

This is the actual data length for the particular data value, that is the size of the object pointed to by `PQgetvalue`. Note that for character-represented values, this size has little to do with the binary size reported by `PQfsize`.

- `PQprint` Prints out all the tuples and, optionally, the attribute names to the specified output stream.

```
void PQprint(FILE* fout,      /* output stream */
             const PGresult *res,
             const PQprintOpt *po);

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;   /* insert to HTML table ... */
```

```
    char    *caption;    /* HTML caption */
    char    **fieldName; /* null terminated array of replacement
 field names */
} PQprintOpt;
```

This function was formerly used by psql to print query results, but this is no longer the case and this function is no longer actively supported.

1.3.6. Retrieving Non-SELECT Result Information

- `PQcmdStatus` Returns the command status string from the SQL command that generated the `PGresult`.

  ```
  char * PQcmdStatus(const PGresult *res);
  ```

- `PQcmdTuples` Returns the number of rows affected by the SQL command.

  ```
  char * PQcmdTuples(const PGresult *res);
  ```

  If the SQL command that generated the `PGresult` was INSERT, UPDATE or DELETE, this returns a string containing the number of rows affected. If the command was anything else, it returns the empty string.

- `PQoidValue` Returns the object ID of the inserted row, if the SQL command was an INSERT that inserted exactly one row into a table that has OIDs. Otherwise, returns `InvalidOid`.

  ```
  Oid PQoidValue(const PGresult *res);
  ```

  The type `Oid` and the constant `InvalidOid` will be defined if you include the libpq header file. They will both be some integer type.

- `PQoidStatus` Returns a string with the object ID of the inserted row, if the SQL command was an INSERT. (The string will be `0` if the INSERT did not insert exactly one row, or if the target table does not have OIDs.) If the command was not an INSERT, returns an empty string.

  ```
  char * PQoidStatus(const PGresult *res);
  ```

  This function is deprecated in favor of `PQoidValue` and is not thread-safe.

1.4. Asynchronous Query Processing

The `PQexec` function is adequate for submitting commands in simple synchronous applications. It has a couple of major deficiencies however:

- `PQexec` waits for the command to be completed. The application may have other work to do (such as maintaining a user interface), in which case it won't want to block waiting for the response.

- Since control is buried inside `PQexec`, it is hard for the frontend to decide it would like to try to cancel the ongoing command. (It can be done from a signal handler, but not otherwise.)

- `PQexec` can return only one `PGresult` structure. If the submitted command string contains multiple SQL commands, all but the last `PGresult` are discarded by `PQexec`.

Applications that do not like these limitations can instead use the underlying functions that `PQexec` is built from: `PQsendQuery` and `PQgetResult`.

Older programs that used this functionality as well as PQputline and PQputnbytes could block waiting to send data to the backend. To address that issue, the function PQsetnonblocking was added.

Old applications can neglect to use PQsetnonblocking and get the older potentially blocking behavior. Newer programs can use PQsetnonblocking to achieve a completely nonblocking connection to the backend.

- PQsetnonblocking Sets the nonblocking status of the connection.

  ```
  int PQsetnonblocking(PGconn *conn, int arg)
  ```

  Sets the state of the connection to nonblocking if *arg* is 1, blocking if *arg* is 0. Returns 0 if OK, -1 if error.

  In the nonblocking state, calls to PQputline, PQputnbytes, PQsendQuery and PQendcopy will not block but instead return an error if they need to be called again.

  When a database connection has been set to nonblocking mode and PQexec is called, it will temporarily set the state of the connection to blocking until the PQexec completes.

  More of libpq is expected to be made safe for PQsetnonblocking functionality in the near future.

- PQisnonblocking Returns the blocking status of the database connection.

  ```
  int PQisnonblocking(const PGconn *conn)
  ```

  Returns 1 if the connection is set to nonblocking mode, 0 if blocking.

- PQsendQuery Submit a command to the server without waiting for the result(s). 1 is returned if the command was successfully dispatched, 0 if not (in which case, use PQerrorMessage to get more information about the failure).

  ```
  int PQsendQuery(PGconn *conn,
                  const char *query);
  ```

  After successfully calling PQsendQuery, call PQgetResult one or more times to obtain the results. PQsendQuery may not be called again (on the same connection) until PQgetResult has returned NULL, indicating that the command is done.

- PQgetResult Wait for the next result from a prior PQsendQuery, and return it. NULL is returned when the query is complete and there will be no more results.

  ```
  PGresult *PQgetResult(PGconn *conn);
  ```

  PQgetResult must be called repeatedly until it returns NULL, indicating that the command is done. (If called when no command is active, PQgetResult will just return NULL at once.) Each non-NULL result from PQgetResult should be processed using the same PGresult accessor functions previously described. Don't forget to free each result object with PQclear when done with it. Note that PQgetResult will block only if a query is active and the necessary response data has not yet been read by PQconsumeInput.

Using PQsendQuery and PQgetResult solves one of PQexec's problems: If a command string contains multiple SQL commands, the results of those commands can be obtained individually. (This allows a simple form of overlapped processing, by the way: the frontend can be handling the results of one query while the backend is still working on later queries in the same command string.) However, calling PQgetResult will still cause the frontend to block until the backend completes the next SQL command. This can be avoided by proper use of three more functions:

- `PQconsumeInput` If input is available from the backend, consume it.

  ```
  int PQconsumeInput(PGconn *conn);
  ```

  `PQconsumeInput` normally returns 1 indicating "no error", but returns 0 if there was some kind of trouble (in which case `PQerrorMessage` is set). Note that the result does not say whether any input data was actually collected. After calling `PQconsumeInput`, the application may check `PQisBusy` and/or `PQnotifies` to see if their state has changed.

  `PQconsumeInput` may be called even if the application is not prepared to deal with a result or notification just yet. The routine will read available data and save it in a buffer, thereby causing a `select()` read-ready indication to go away. The application can thus use `PQconsumeInput` to clear the `select()` condition immediately, and then examine the results at leisure.

- `PQisBusy` Returns 1 if a query is busy, that is, `PQgetResult` would block waiting for input. A 0 return indicates that `PQgetResult` can be called with assurance of not blocking.

  ```
  int PQisBusy(PGconn *conn);
  ```

  `PQisBusy` will not itself attempt to read data from the backend; therefore `PQconsumeInput` must be invoked first, or the busy state will never end.

- `PQflush` Attempt to flush any data queued to the backend, returns 0 if successful (or if the send queue is empty) or `EOF` if it failed for some reason.

  ```
  int PQflush(PGconn *conn);
  ```

  `PQflush` needs to be called on a nonblocking connection before calling `select()` to determine if a response has arrived. If 0 is returned it ensures that there is no data queued to the backend that has not actually been sent. Only applications that have used `PQsetnonblocking` have a need for this.

- `PQsocket` Obtain the file descriptor number for the backend connection socket. A valid descriptor will be >= 0; a result of -1 indicates that no backend connection is currently open.

  ```
  int PQsocket(const PGconn *conn);
  ```

  `PQsocket` should be used to obtain the backend socket descriptor in preparation for executing `select()`. This allows an application using a blocking connection to wait for either backend responses or other conditions. If the result of `select()` indicates that data can be read from the backend socket, then `PQconsumeInput` should be called to read the data; after which, `PQisBusy`, `PQgetResult`, and/or `PQnotifies` can be used to process the response.

  Nonblocking connections (that have used `PQsetnonblocking`) should not use `select()` until `PQflush` has returned 0 indicating that there is no buffered data waiting to be sent to the backend.

A typical frontend using these functions will have a main loop that uses `select` to wait for all the conditions that it must respond to. One of the conditions will be input available from the backend, which in `select`'s terms is readable data on the file descriptor identified by `PQsocket`. When the main loop detects input ready, it should call `PQconsumeInput` to read the input. It can then call `PQisBusy`, followed by `PQgetResult` if `PQisBusy` returns false (0). It can also call `PQnotifies` to detect NOTIFY messages (see Section 1.6, "Asynchronous Notification").

A frontend that uses `PQsendQuery`/`PQgetResult` can also attempt to cancel a command that is still being processed by the backend.

- `PQrequestCancel` Request that PostgreSQL abandon processing of the current command.

```
int PQrequestCancel(PGconn *conn);
```

The return value is 1 if the cancel request was successfully dispatched, 0 if not. (If not, `PQerrorMessage` tells why not.) Successful dispatch is no guarantee that the request will have any effect, however. Regardless of the return value of `PQrequestCancel`, the application must continue with the normal result-reading sequence using `PQgetResult`. If the cancellation is effective, the current command will terminate early and return an error result. If the cancellation fails (say, because the backend was already done processing the command), then there will be no visible result at all.

Note that if the current command is part of a transaction, cancellation will abort the whole transaction.

`PQrequestCancel` can safely be invoked from a signal handler. So, it is also possible to use it in conjunction with plain `PQexec`, if the decision to cancel can be made in a signal handler. For example, psql invokes `PQrequestCancel` from a SIGINT signal handler, thus allowing interactive cancellation of queries that it issues through `PQexec`. Note that `PQrequestCancel` will have no effect if the connection is not currently open or the backend is not currently processing a command.

1.5. The Fast-Path Interface

PostgreSQL provides a fast-path interface to send function calls to the backend. This is a trapdoor into system internals and can be a potential security hole. Most users will not need this feature.

- `PQfn` Request execution of a backend function via the fast-path interface.

```
PGresult* PQfn(PGconn* conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);
```

The *`fnid`* argument is the object identifier of the function to be executed. *`result_buf`* is the buffer in which to place the return value. The caller must have allocated sufficient space to store the return value (there is no check!). The actual result length will be returned in the integer pointed to by *`result_len`*. If a 4-byte integer result is expected, set *`result_is_int`* to 1; otherwise set it to 0. (Setting *`result_is_int`* to 1 tells libpq to byte-swap the value if necessary, so that it is delivered as a proper int value for the client machine. When *`result_is_int`* is 0, the byte string sent by the backend is returned unmodified.) *`args`* and *`nargs`* specify the arguments to be passed to the function.

```
typedef struct {
    int len;
    int isint;
    union {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;
```

`PQfn` always returns a valid `PGresult*`. The resultStatus should be checked before the result is used. The caller is responsible for freeing the `PGresult` with `PQclear` when it is no longer needed.

1.6. Asynchronous Notification

PostgreSQL supports asynchronous notification via the LISTEN and NOTIFY commands. A backend registers its interest in a particular notification condition with the LISTEN command (and can stop listening with the UNLISTEN command). All backends listening on a particular condition will be notified asynchronously when a NOTIFY of that condition name is executed by any backend. No additional information is passed from the notifier to the listener. Thus, typically, any actual data that needs to be communicated is transferred through a database relation. Commonly the condition name is the same as the associated relation, but it is not necessary for there to be any associated relation.

libpq applications submit LISTEN and UNLISTEN commands as ordinary SQL command. Subsequently, arrival of NOTIFY messages can be detected by calling PQnotifies.

- PQnotifies Returns the next notification from a list of unhandled notification messages received from the backend. Returns NULL if there are no pending notifications. Once a notification is returned from PQnotifies, it is considered handled and will be removed from the list of notifications.

  ```
  PGnotify* PQnotifies(PGconn *conn);

  typedef struct pgNotify {
      char relname[NAMEDATALEN];       /* name of relation
                                        * containing data */
      int  be_pid;                     /* process id of backend */
  } PGnotify;
  ```

  After processing a PGnotify object returned by PQnotifies, be sure to free it with free() to avoid a memory leak.

  Note

  In PostgreSQL 6.4 and later, the be_pid is that of the notifying backend, whereas in earlier versions it was always the PID of your own backend.

The second sample program gives an example of the use of asynchronous notification.

PQnotifies() does not actually read backend data; it just returns messages previously absorbed by another libpq function. In prior releases of libpq, the only way to ensure timely receipt of NOTIFY messages was to constantly submit queries, even empty ones, and then check PQnotifies() after each PQexec(). While this still works, it is deprecated as a waste of processing power.

A better way to check for NOTIFY messages when you have no useful queries to make is to call PQconsumeInput(), then check PQnotifies(). You can use select() to wait for backend data to arrive, thereby using no CPU power unless there is something to do. (See PQsocket() to obtain the file descriptor number to use with select().) Note that this will work OK whether you submit queries with PQsendQuery/PQgetResult or simply use PQexec. You should, however, remember to check PQnotifies() after each PQgetResult or PQexec, to see if any notifications came in during the processing of the query.

1.7. Functions Associated with the COPY Command

The COPY command in PostgreSQL has options to read from or write to the network connection used by libpq. Therefore, functions are necessary to access this network connection directly so applications may take advantage of this capability.

These functions should be executed only after obtaining a PGRES_COPY_OUT or PGRES_COPY_IN result object from PQexec or PQgetResult.

- PQgetline Reads a newline-terminated line of characters (transmitted by the backend server) into a buffer string of size length.

```
int PQgetline(PGconn *conn,
              char *string,
              int length)
```

  Like fgets, this routine copies up to length-1 characters into string. It is like gets, however, in that it converts the terminating newline into a zero byte. PQgetline returns EOF at the end of input, 0 if the entire line has been read, and 1 if the buffer is full but the terminating newline has not yet been read.

  Notice that the application must check to see if a new line consists of the two characters \., which indicates that the backend server has finished sending the results of the copy command. If the application might receive lines that are more than length-1 characters long, care is needed to be sure one recognizes the \. line correctly (and does not, for example, mistake the end of a long data line for a terminator line). The code in src/bin/psql/copy.c contains example routines that correctly handle the copy protocol.

- PQgetlineAsync Reads a newline-terminated line of characters (transmitted by the backend server) into a buffer without blocking.

```
int PQgetlineAsync(PGconn *conn,
                   char *buffer,
                   int bufsize)
```

  This routine is similar to PQgetline, but it can be used by applications that must read COPY data asynchronously, that is without blocking. Having issued the COPY command and gotten a PGRES_COPY_OUT response, the application should call PQconsumeInput and PQgetlineAsync until the end-of-data signal is detected. Unlike PQgetline, this routine takes responsibility for detecting end-of-data. On each call, PQgetlineAsync will return data if a complete newline- terminated data line is available in libpq's input buffer, or if the incoming data line is too long to fit in the buffer offered by the caller. Otherwise, no data is returned until the rest of the line arrives.

  The routine returns -1 if the end-of-copy-data marker has been recognized, or 0 if no data is available, or a positive number giving the number of bytes of data returned. If -1 is returned, the caller must next call PQendcopy, and then return to normal processing. The data returned will not extend beyond a newline character. If possible a whole line will be returned at one time. But if the buffer offered by the caller is too small to hold a line sent by the backend, then a partial data line will be returned. This can be detected by testing whether the last returned byte is \n or not. The returned string is not null-terminated. (If you want to add a terminating null, be sure to pass a *bufsize* one smaller than the room actually available.)

- PQputline Sends a null-terminated string to the backend server. Returns 0 if OK, EOF if unable to send the string.

```
int PQputline(PGconn *conn,
              const char *string);
```

  Note the application must explicitly send the two characters \. on a final line to indicate to the backend that it has finished sending its data.

- PQputnbytes Sends a non-null-terminated string to the backend server. Returns 0 if OK, EOF if unable to send the string.

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

This is exactly like `PQputline`, except that the data buffer need not be null-terminated since the number of bytes to send is specified directly.

- `PQendcopy` Synchronizes with the backend. This function waits until the backend has finished the copy. It should either be issued when the last string has been sent to the backend using `PQputline` or when the last string has been received from the backend using `PGgetline`. It must be issued or the backend may get "out of sync" with the frontend. Upon return from this function, the backend is ready to receive the next SQL command. The return value is 0 on successful completion, nonzero otherwise.

  ```
  int PQendcopy(PGconn *conn);
  ```

  As an example:

  ```
  PQexec(conn, "CREATE TABLE foo (a int4, b char(16), d double
   precision)");
  PQexec(conn, "COPY foo FROM STDIN");
  PQputline(conn, "3\thello world\t4.5\n");
  PQputline(conn,"4\tgoodbye world\t7.11\n");
  ...
  PQputline(conn,"\\.\n");
  PQendcopy(conn);
  ```

When using `PQgetResult`, the application should respond to a `PGRES_COPY_OUT` result by executing `PQgetline` repeatedly, followed by `PQendcopy` after the terminator line is seen. It should then return to the `PQgetResult` loop until `PQgetResult` returns NULL. Similarly a `PGRES_COPY_IN` result is processed by a series of `PQputline` calls followed by `PQendcopy`, then return to the `PQgetResult` loop. This arrangement will ensure that a copy in or copy out command embedded in a series of SQL commands will be executed correctly.

Older applications are likely to submit a copy in or copy out via `PQexec` and assume that the transaction is done after `PQendcopy`. This will work correctly only if the copy in/out is the only SQL command in the command string.

1.8. libpq Tracing Functions

- `PQtrace` Enable tracing of the frontend/backend communication to a debugging file stream.

  ```
  void PQtrace(PGconn *conn
               FILE *debug_port)
  ```

- `PQuntrace` Disable tracing started by `PQtrace`.

  ```
  void PQuntrace(PGconn *conn)
  ```

1.9. libpq Control Functions

- `PQsetNoticeProcessor` Control reporting of notice and warning messages generated by libpq.

  ```
  typedef void (*PQnoticeProcessor) (void *arg, const char *message);
  ```

```
PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

By default, libpq prints notice messages from the backend on `stderr`, as well as a few error messages that it generates by itself. This behavior can be overridden by supplying a callback function that does something else with the messages. The callback function is passed the text of the error message (which includes a trailing newline), plus a void pointer that is the same one passed to `PQsetNoticeProcessor`. (This pointer can be used to access application-specific state if needed.) The default notice processor is simply

```
static void
defaultNoticeProcessor(void * arg, const char * message)
{
    fprintf(stderr, "%s", message);
}
```

To use a special notice processor, call `PQsetNoticeProcessor` just after creation of a new `PGconn` object.

The return value is the pointer to the previous notice processor. If you supply a callback function pointer of NULL, no action is taken, but the current pointer is returned.

Once you have set a notice processor, you should expect that that function could be called as long as either the `PGconn` object or `PGresult` objects made from it exist. At creation of a `PGresult`, the `PGconn`'s current notice processor pointer is copied into the `PGresult` for possible use by routines like `PQgetvalue`.

1.10. Environment Variables

The following environment variables can be used to select default connection parameter values, which will be used by `PQconnectdb` or `PQsetdbLogin` if no value is directly specified by the calling code. These are useful to avoid hard-coding database names into simple application programs.

- `PGHOST` sets the default server name. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is the name of the directory in which the socket file is stored (default `/tmp`).
- `PGPORT` sets the default TCP port number or Unix-domain socket file extension for communicating with the PostgreSQL backend.
- `PGDATABASE` sets the default PostgreSQL database name.
- `PGUSER` sets the user name used to connect to the database and for authentication.
- `PGPASSWORD` sets the password used if the backend demands password authentication. This is not recommended because the password can be read by others using the `ps` command with special options on some platforms.
- `PGREALM` sets the Kerberos realm to use with PostgreSQL, if it is different from the local realm. If `PGREALM` is set, PostgreSQL applications will attempt authentication with servers for this realm and use separate ticket files to avoid conflicts with local ticket files. This environment variable is only used if Kerberos authentication is selected by the backend.
- `PGOPTIONS` sets additional runtime options for the PostgreSQL backend.

- PGTTY sets the file or tty on which debugging messages from the backend server are displayed.

The following environment variables can be used to specify user-level default behavior for every PostgreSQL session:

- PGDATESTYLE sets the default style of date/time representation.
- PGTZ sets the default time zone.
- PGCLIENTENCODING sets the default client encoding (if multibyte support was selected when configuring PostgreSQL).

The following environment variables can be used to specify default internal behavior for every PostgreSQL session:

- PGGEQO sets the default mode for the genetic optimizer.

Refer to the SET SQL command for information on correct values for these environment variables.

1.11. Threading Behavior

libpq is thread-safe as of PostgreSQL 7.0, so long as no two threads attempt to manipulate the same PGconn object at the same time. In particular, you cannot issue concurrent queries from different threads through the same connection object. (If you need to run concurrent queries, start up multiple connections.)

PGresult objects are read-only after creation, and so can be passed around freely between threads.

The deprecated functions PQoidStatus and fe_setauthsvc are not thread-safe and should not be used in multithread programs. PQoidStatus can be replaced by PQoidValue. There is no good reason to call fe_setauthsvc at all.

Libpq clients using the crypt encryption method rely on the crypt() operating system function, which is often not thread-safe. It is better to use MD5 encryption, which is thread-safe on all platforms.

1.12. Building Libpq Programs

To build (i.e., compile and link) your libpq programs you need to do all of the following things:

- Include the libpq-fe.h header file:

  ```
  #include <libpq-fe.h>
  ```

  If you failed to do that then you will normally get error messages from your compiler similar to

  ```
  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)
  ```

- Point your compiler to the directory where the PostgreSQL header files were installed, by supplying the -I*directory* option to your compiler. (In some cases the compiler will look into the directory in question by default, so you can omit this option.) For instance, your compile command line could look like:

```
cc -c -I/usr/local/pgsql/include testprog.c
```

If you are using makefiles then add the option to the `CPPFLAGS` variable:

```
CPPFLAGS += -I/usr/local/pgsql/include
```

If there is any chance that your program might be compiled by other users then you should not hardcode the directory location like that. Instead, you can run the utility `pg_config` to find out where the header files are on the local system:

```
$ pg_config --includedir
/usr/local/include
```

Failure to specify the correct option to the compiler will result in an error message such as

```
testlibpq.c:8:22: libpq-fe.h: No such file or directory
```

- When linking the final program, specify the option `-lpq` so that the libpq library gets pulled in, as well as the option `-L`*directory* to point it to the directory where the libpq library resides. (Again, the compiler will search some directories by default.) For maximum portability, put the `-L` option before the `-lpq` option. For example:

  ```
  cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq
  ```

  You can find out the library directory using `pg_config` as well:

  ```
  $ pg_config --libdir
  /usr/local/pgsql/lib
  ```

  Error messages that point to problems in this area could look like the following.

  ```
  testlibpq.o: In function `main':
  testlibpq.o(.text+0x60): undefined reference to `PQsetdbLogin'
  testlibpq.o(.text+0x71): undefined reference to `PQstatus'
  testlibpq.o(.text+0xa4): undefined reference to `PQerrorMessage'
  ```

  This means you forgot `-lpq`.

  ```
  /usr/bin/ld: cannot find -lpq
  ```

  This means you forgot the `-L` or did not specify the right path.

If your codes references the header file `libpq-int.h` and you refuse to fix your code to not use it, starting in PostgreSQL 7.2, this file will be found in *includedir*`/postgresql/internal/libpq-int.h`, so you need to add the appropriate `-I` option to your compiler command line.

1.13. Example Programs

Example 1.1. libpq Example Program 1

```
/*
 * testlibpq.c
 *
 * Test the C version of libpq, the PostgreSQL frontend
 * library.
```

```
 */
#include <stdio.h>
#include <libpq-fe.h>

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;

    /* FILE *debug; */

    PGconn     *conn;
    PGresult   *res;

    /*
     * begin, by setting the parameters for a backend connection if
 the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the
 backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend
 server */
    dbName = "template1";

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n",
 dbName);
        fprintf(stderr, "%s", PQerrorMessage(conn));
```

```
        exit_nicely(conn);
    }

    /* debug = fopen("/tmp/trace.out","w"); */
    /* PQtrace(conn, debug);  */

    /* start a transaction block */
    res = PQexec(conn, "BEGIN");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to
avoid
     * memory leaks
     */
    PQclear(res);

    /*
     * fetch rows from the pg_database, the system catalog of
     * databases
     */
    res = PQexec(conn, "DECLARE mycursor CURSOR FOR SELECT * FROM
pg_database");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);
    res = PQexec(conn, "FETCH ALL in mycursor");
    if (!res || PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL command didn't return tuples
properly\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /* first, print out the attribute names */
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");

    /* next, print out the rows */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
```

```
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }
    PQclear(res);

    /* close the cursor */
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);

    /* commit the transaction */
    res = PQexec(conn, "COMMIT");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    /* fclose(debug); */
    return 0;

}
```

Example 1.2. libpq Example Program 2

```
/*
 * testlibpq2.c
 *  Test of the asynchronous notification interface
 *
 * Start this program, then from psql in another window do
 *   NOTIFY TBL2;
 *
 * Or, if you want to get fancy, try this:
 * Populate a database with the following:
 *
 *   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);
 *
 * and do
 *
 *   INSERT INTO TBL1 values (10);
 *
 */
#include <stdio.h>
#include "libpq-fe.h"

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}
```

```
main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;

    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;

    /*
     * begin, by setting the parameters for a backend connection if
 the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the
 backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend
 server */
    dbName = getenv("USER");    /* change this to the name of your
 test
                                 * database */

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n",
 dbName);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "LISTEN TBL2");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "LISTEN command failed\n");
        PQclear(res);
        exit_nicely(conn);
```

```
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to
 avoid
     * memory leaks
     */
    PQclear(res);

    while (1)
    {

        /*
         * wait a little bit between checks; waiting with select()
         * would be more efficient.
         */
        sleep(1);
        /* collect any asynchronous backend messages */
        PQconsumeInput(conn);
        /* check for asynchronous notify messages */
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                 "ASYNC NOTIFY of '%s' from backend pid '%d' received
\n",
                    notify->relname, notify->be_pid);
            free(notify);
        }
    }

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Example 1.3. libpq Example Program 3

```
/*
 * testlibpq3.c Test the C version of Libpq, the PostgreSQL frontend
 * library. tests the binary cursor interface
 *
 *
 *
 * populate a database by doing the following:
 *
 * CREATE TABLE test1 (i int4, d real, p polygon);
 *
 * INSERT INTO test1 values (1, 3.567, polygon '(3.0, 4.0, 1.0,
 2.0)');
 *
 * INSERT INTO test1 values (2, 89.05, polygon '(4.0, 3.0, 2.0,
 1.0)');
 *
```

```
 * the expected output is:
 *
 * tuple 0: got i = (4 bytes) 1, d = (4 bytes) 3.567000, p = (4
 * bytes) 2 points   boundbox = (hi=3.000000/4.000000, lo =
 * 1.000000,2.000000) tuple 1: got i = (4 bytes) 2, d = (4 bytes)
 * 89.050003, p = (4 bytes) 2 points   boundbox =
 * (hi=4.000000/3.000000, lo = 2.000000,1.000000)
 *
 *
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "utils/geo_decls.h"    /* for the POLYGON type */

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;
    int         i_fnum,
                d_fnum,
                p_fnum;
    PGconn     *conn;
    PGresult   *res;

    /*
     * begin, by setting the parameters for a backend connection if
 the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the
 backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend
 server */

    dbName = getenv("USER");    /* change this to the name of your
 test
```

```
                                * database */

   /* make a connection to the database */
   conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

   /*
    * check to see that the backend connection was successfully made
    */
   if (PQstatus(conn) == CONNECTION_BAD)
   {
       fprintf(stderr, "Connection to database '%s' failed.\n",
dbName);
       fprintf(stderr, "%s", PQerrorMessage(conn));
       exit_nicely(conn);
   }

   /* start a transaction block */
   res = PQexec(conn, "BEGIN");
   if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
   {
       fprintf(stderr, "BEGIN command failed\n");
       PQclear(res);
       exit_nicely(conn);
   }

   /*
    * should PQclear PGresult whenever it is no longer needed to
avoid
    * memory leaks
    */
   PQclear(res);

   /*
    * fetch rows from the pg_database, the system catalog of
    * databases
    */
   res = PQexec(conn, "DECLARE mycursor BINARY CURSOR FOR SELECT *
FROM test1");
   if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
   {
       fprintf(stderr, "DECLARE CURSOR command failed\n");
       PQclear(res);
       exit_nicely(conn);
   }
   PQclear(res);

   res = PQexec(conn, "FETCH ALL in mycursor");
   if (!res || PQresultStatus(res) != PGRES_TUPLES_OK)
   {
       fprintf(stderr, "FETCH ALL command didn't return tuples
properly\n");
       PQclear(res);
       exit_nicely(conn);
   }
```

```
    i_fnum = PQfnumber(res, "i");
    d_fnum = PQfnumber(res, "d");
    p_fnum = PQfnumber(res, "p");

    for (i = 0; i < 3; i++)
    {
        printf("type[%d] = %d, size[%d] = %d\n",
               i, PQftype(res, i),
               i, PQfsize(res, i));
    }
    for (i = 0; i < PQntuples(res); i++)
    {
        int        *ival;
        float      *dval;
        int         plen;
        POLYGON    *pval;

        /* we hard-wire this to the 3 fields we know about */
        ival = (int *) PQgetvalue(res, i, i_fnum);
        dval = (float *) PQgetvalue(res, i, d_fnum);
        plen = PQgetlength(res, i, p_fnum);

        /*
         * plen doesn't include the length field so need to
         * increment by VARHDSZ
         */
        pval = (POLYGON *) malloc(plen + VARHDRSZ);
        pval->size = plen;
        memmove((char *) &pval->npts, PQgetvalue(res, i, p_fnum),
plen);
        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d,\n",
               PQgetlength(res, i, i_fnum), *ival);
        printf(" d = (%d bytes) %f,\n",
               PQgetlength(res, i, d_fnum), *dval);
        printf(" p = (%d bytes) %d points \tboundbox = (hi=%f/%f, lo =
%f,%f)\n",
               PQgetlength(res, i, d_fnum),
               pval->npts,
               pval->boundbox.xh,
               pval->boundbox.yh,
               pval->boundbox.xl,
               pval->boundbox.yl);
    }
    PQclear(res);

    /* close the cursor */
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);

    /* commit the transaction */
    res = PQexec(conn, "COMMIT");
    PQclear(res);
```

```
    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Chapter 2. Large Objects

2.1. Introduction

In PostgreSQL releases prior to 7.1, the size of any row in the database could not exceed the size of a data page. Since the size of a data page is 8192 bytes (the default, which can be raised up to 32768), the upper limit on the size of a data value was relatively low. To support the storage of larger atomic values, PostgreSQL provided and continues to provide a large object interface. This interface provides file-oriented access to user data that has been declared to be a large object.

POSTGRES 4.2, the indirect predecessor of PostgreSQL, supported three standard implementations of large objects: as files external to the POSTGRES server, as external files managed by the POSTGRES server, and as data stored within the POSTGRES database. This caused considerable confusion among users. As a result, only support for large objects as data stored within the database is retained in PostgreSQL. Even though this is slower to access, it provides stricter data integrity. For historical reasons, this storage scheme is referred to as *Inversion large objects*. (You will see the term Inversion used occasionally to mean the same thing as large object.) Since PostgreSQL 7.1, all large objects are placed in one system table called `pg_largeobject`.

PostgreSQL 7.1 introduced a mechanism (nicknamed "TOAST") that allows data rows to be much larger than individual data pages. This makes the large object interface partially obsolete. One remaining advantage of the large object interface is that it allows random access to the data, i.e., the ability to read or write small chunks of a large value. It is planned to equip TOAST with such functionality in the future.

This section describes the implementation and the programming and query language interfaces to PostgreSQL large object data. We use the libpq C library for the examples in this section, but most programming interfaces native to PostgreSQL support equivalent functionality. Other interfaces may use the large object interface internally to provide generic support for large values. This is not described here.

2.2. Implementation Features

The large object implementation breaks large objects up into "chunks" and stores the chunks in tuples in the database. A B-tree index guarantees fast searches for the correct chunk number when doing random access reads and writes.

2.3. Interfaces

The facilities PostgreSQL provides to access large objects, both in the backend as part of user-defined functions or the front end as part of an application using the interface, are described below. For users familiar with POSTGRES 4.2, PostgreSQL has a new set of functions providing a more coherent interface.

Note

All large object manipulation *must* take place within an SQL transaction. This requirement is strictly enforced as of PostgreSQL 6.5, though it has been an implicit requirement in previous versions, resulting in misbehavior if ignored.

The PostgreSQL large object interface is modeled after the Unix file-system interface, with analogues of `open(2)`, `read(2)`, `write(2)`, `lseek(2)`, etc. User functions call these routines to retrieve only the data of interest from a large object. For example, if a large object type called `mugshot` existed that stored photographs of faces, then a function called `beard` could be declared on `mugshot` data. `beard`

could look at the lower third of a photograph, and determine the color of the beard that appeared there, if any. The entire large-object value need not be buffered, or even examined, by the `beard` function. Large objects may be accessed from dynamically-loaded C functions or database client programs that link the library. PostgreSQL provides a set of routines that support opening, reading, writing, closing, and seeking on large objects.

2.3.1. Creating a Large Object

The routine

```
Oid lo_creat(PGconn *conn, int mode)
```

creates a new large object. *mode* is a bit mask describing several different attributes of the new object. The symbolic constants listed here are defined in the header file `libpq/libpq-fs.h`. The access type (read, write, or both) is controlled by or'ing together the bits `INV_READ` and `INV_WRITE`. The low-order sixteen bits of the mask have historically been used at Berkeley to designate the storage manager number on which the large object should reside. These bits should always be zero now. The commands below create a large object:

```
inv_oid = lo_creat(INV_READ|INV_WRITE);
```

2.3.2. Importing a Large Object

To import an operating system file as a large object, call

```
Oid lo_import(PGconn *conn, const char *filename)
```

filename specifies the operating system name of the file to be imported as a large object.

2.3.3. Exporting a Large Object

To export a large object into an operating system file, call

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename)
```

The *lobjId* argument specifies the OID of the large object to export and the *filename* argument specifies the operating system name name of the file.

2.3.4. Opening an Existing Large Object

To open an existing large object, call

```
int lo_open(PGconn *conn, Oid lobjId, int mode)
```

The *lobjId* argument specifies the OID of the large object to open. The *mode* bits control whether the object is opened for reading (`INV_READ`), writing (`INV_WRITE`), or both. A large object cannot be opened before it is created. `lo_open` returns a large object descriptor for later use in `lo_read`, `lo_write`, `lo_lseek`, `lo_tell`, and `lo_close`.

2.3.5. Writing Data to a Large Object

The routine

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len)
```

writes *len* bytes from *buf* to large object *fd*. The *fd* argument must have been returned by a previous lo_open. The number of bytes actually written is returned. In the event of an error, the return value is negative.

2.3.6. Reading Data from a Large Object

The routine

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len)
```

reads *len* bytes from large object *fd* into *buf*. The *fd* argument must have been returned by a previous lo_open. The number of bytes actually read is returned. In the event of an error, the return value is negative.

2.3.7. Seeking on a Large Object

To change the current read or write location on a large object, call

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence)
```

This routine moves the current location pointer for the large object described by *fd* to the new location specified by *offset*. The valid values for *whence* are SEEK_SET, SEEK_CUR, and SEEK_END.

2.3.8. Closing a Large Object Descriptor

A large object may be closed by calling

```
int lo_close(PGconn *conn, int fd)
```

where *fd* is a large object descriptor returned by lo_open. On success, lo_close returns zero. On error, the return value is negative.

2.3.9. Removing a Large Object

To remove a large object from the database, call

```
Oid lo_unlink(PGconn *conn, Oid lobjId)
```

The *lobjId* argument specifies the OID of the large object to remove.

2.4. Server-side Built-in Functions

There are two built-in registered functions, lo_import and lo_export which are convenient for use in SQL queries. Here is an example of their use

```
CREATE TABLE image (
    name            text,
    raster          oid
);

INSERT INTO image (name, raster)
    VALUES ('beautiful image', lo_import('/etc/motd'));
```

```
SELECT lo_export(image.raster, '/tmp/motd') FROM image
    WHERE name = 'beautiful image';
```

2.5. Accessing Large Objects from Libpq

Example 2.1, “Large Objects with Libpq Example Program” is a sample program which shows how the large object interface in libpq can be used. Parts of the program are commented out but are left in the source for the reader's benefit. This program can be found in `src/test/examples/testlo.c` in the source distribution. Frontend applications which use the large object interface in libpq should include the header file `libpq/libpq-fs.h` and link with the libpq library.

Example 2.1. Large Objects with Libpq Example Program

```
/*--------------------------------------------------------------
 *
 * testlo.c--
 *    test using large objects with libpq
 *
 * Copyright (c) 1994, Regents of the University of California
 *
 *--------------------------------------------------------------
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE          1024

/*
 * importFile
 *    import file "in_filename" into database as large object
 "lobjOid"
 *
 */
Oid
importFile(PGconn *conn, char *filename)
{
    Oid         lobjId;
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * open the file to be read in
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "can't open unix file %s\n", filename);
    }
```

```
    /*
     * create the large object
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "can't create large object\n");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading large object\n");
    }

    (void) close(fd);
    (void) lo_close(conn, lobj_fd);

    return lobjId;
}

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, "can't open large object %d\n",
                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] = ' ';
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
```

```
}

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, "can't open large object %d\n",
                lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = ' ';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len -
 nwritten);
        nwritten += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

/*
 * exportFile *    export large object "lobjOid" to file
 "out_filename"
 *
 */
void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * create an inversion "object"
     */
```

```
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }

    /*
     * open the file to be written to
     */
    fd = open(filename, O_CREAT | O_WRONLY, 0666);
    if (fd < 0)
    {                               /* error */
        fprintf(stderr, "can't open unix file %s\n",
                filename);
    }

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing %s\n",
                    filename);
        }
    }

    (void) lo_close(conn, lobj_fd);
    (void) close(fd);

    return;
}

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, "Usage: %s database_name in_filename
 out_filename\n",
                argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * set up the connection
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* check to see that the backend connection was successfully made
 */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n",
 database);
        fprintf(stderr, "%s", 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);
/*
    printf("as large object %d.\n", lobjOid);

    printf("picking out bytes 1000-2000 of the large object\n");
    pickout(conn, lobjOid, 1000, 1000);

    printf("overwriting bytes 1000-2000 of the large object with X's
\n");
    overwrite(conn, lobjOid, 1000, 1000);
*/

    printf("exporting large object to file %s\n", out_filename);
/*    exportFile(conn, lobjOid, out_filename); */
    lo_export(conn, lobjOid, out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
```

Chapter 3. libpq++ - C++ Binding Library

3.1. Introduction

libpq++ is the C++ API to PostgreSQL. libpq++ is a set of classes that allow client programs to connect to the PostgreSQL backend server. These connections come in two forms: a database class and a large object class.

The database class is intended for manipulating a database. You can send all sorts of SQL queries and commands to the PostgreSQL backend server and retrieve the responses of the server.

The large object class is intended for manipulating a large object in a database. Although a large object instance can send normal queries to the PostgreSQL backend server it is only intended for simple queries that do not return any data. A large object should be seen as a file stream. In the future it should behave much like the C++ file streams `cin`, `cout` and `cerr`.

This chapter is based on the documentation for the libpq C library (see Chapter 1, *libpq - C Library*). There are several examples of libpq++ applications in `src/interfaces/libpq++/examples` in the source distribution.

3.2. Control and Initialization

3.2.1. Environment Variables

The following environment variables can be used to set up default values for an environment and to avoid hard-coding database names into an application program:

> **Note**
>
> Refer to Section 1.10, “Environment Variables” for a complete list of available connection options.

The following environment variables can be used to select default connection parameter values, which will be used by `PQconnectdb` or `PQsetdbLogin` if no value is directly specified by the calling code. These are useful to avoid hard-coding database names into simple application programs.

> **Note**
>
> `libpq++` uses only environment variables or `libpq`'s `PQconnectdb` *`conninfo`* style strings.

- `PGHOST` sets the default server name. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is the name of the directory in which the socket file is stored (default `/tmp`).
- `PGPORT` sets the default TCP port number or Unix-domain socket file extension for communicating with the PostgreSQL backend.

- PGDATABASE sets the default PostgreSQL database name.
- PGUSER sets the user name used to connect to the database and for authentication.
- PGPASSWORD sets the password used if the backend demands password authentication. This is not recommended because the password can be read by others using the ps command with special options on some platforms.
- PGREALM sets the Kerberos realm to use with PostgreSQL, if it is different from the local realm. If PGREALM is set, PostgreSQL applications will attempt authentication with servers for this realm and use separate ticket files to avoid conflicts with local ticket files. This environment variable is only used if Kerberos authentication is selected by the backend.
- PGOPTIONS sets additional runtime options for the PostgreSQL backend.
- PGTTY sets the file or tty on which debugging messages from the backend server are displayed.

The following environment variables can be used to specify user-level default behavior for every PostgreSQL session:

- PGDATESTYLE sets the default style of date/time representation.
- PGTZ sets the default time zone.

The following environment variables can be used to specify default internal behavior for every PostgreSQL session:

- PGGEQO sets the default mode for the genetic optimizer.

Refer to the SET SQL command for information on correct values for these environment variables.

3.3. libpq++ Classes

3.3.1. Connection Class: PgConnection

The connection class makes the actual connection to the database and is inherited by all of the access classes.

3.3.2. Database Class: PgDatabase

The database class provides C++ objects that have a connection to a backend server. To create such an object one first needs the appropriate environment for the backend to access. The following constructors deal with making a connection to a backend server from a C++ program.

3.4. Database Connection Functions

- PgConnection makes a new connection to a backend database server.

  ```
  PgConnection::PgConnection(const char *conninfo)
  ```

  The *conninfo* string is the same as for the underlying libpq PQconnectdb function.

Although typically called from one of the access classes, a connection to a backend server is possible by creating a `PgConnection` object.

- `ConnectionBad` returns whether or not the connection to the backend server succeeded or failed.

  ```
  bool PgConnection::ConnectionBad() const
  ```

  Returns true if the connection failed.

- `Status` returns the status of the connection to the backend server.

  ```
  ConnStatusType PgConnection::Status()
  ```

  Returns either `CONNECTION_OK` or `CONNECTION_BAD` depending on the state of the connection.

- `PgDatabase` makes a new connection to a backend database server.

  ```
  PgDatabase(const char *conninfo)
  ```

  After a `PgDatabase` has been created it should be checked to make sure the connection to the database succeeded before sending queries to the object. This can easily be done by retrieving the current status of the `PgDatabase` object with the `Status` or `ConnectionBad` methods.

- `DBName` returns the name of the current database.

  ```
  const char *PgConnection::DBName()
  ```

- `Notifies` returns the next notification from a list of unhandled notification messages received from the backend.

  ```
  PGnotify* PgConnection::Notifies()
  ```

  See `PQnotifies` in libpq for details.

3.5. Query Execution Functions

3.5.1. Main Routines

- `Exec` sends a command to the backend server. It's probably more desirable to use one of the next two functions.

  ```
  ExecStatusType PgConnection::Exec(const char* query)
  ```

  Returns the result status of the command. The following status results can be expected:

  `PGRES_EMPTY_QUERY`
  `PGRES_COMMAND_OK`, if the command was not a query
  `PGRES_TUPLES_OK`, if the query successfully returned tuples
  `PGRES_COPY_OUT`
  `PGRES_COPY_IN`
  `PGRES_BAD_RESPONSE`, if an unexpected response was received
  `PGRES_NONFATAL_ERROR`
  `PGRES_FATAL_ERROR`

- `ExecCommandOk` sends a non-query command (one that does not return rows) to the backend server.

```
int PgConnection::ExecCommandOk(const char *query)
```

Returns true (1) if the command succeeds.

- `ExecTuplesOk` Sends a query command (one that returns rows) to the backend server.

```
int PgConnection::ExecTuplesOk(const char *query)
```

Returns true (1) if the query succeeds.

- `ErrorMessage` returns the last error message text.

```
const char *PgConnection::ErrorMessage()
```

3.5.2. Retrieving SELECT Result Information

- `Tuples` returns the number of tuples (rows) in the query result.

```
int PgDatabase::Tuples() const
```

- `Fields` returns the number of fields (rows) in each tuple of the query result.

```
int PgDatabase::Fields()
```

- `FieldName` returns the field (column) name associated with the given field index. Field indices start at 0.

```
const char *PgDatabase::FieldName(int field_num) const
```

- `FieldNum` returns the field (column) index associated with the given field name.

```
int PgDatabase::FieldNum(const char* field_name) const
```

-1 is returned if the given name does not match any field.

- `FieldType` returns the field type associated with the given field index. The integer returned is an internal coding of the type. Field indices start at 0.

```
Oid PgDatabase::FieldType(int field_num) const
```

- `FieldType` returns the field type associated with the given field name. The integer returned is an internal coding of the type. Field indices start at 0.

```
Oid PgDatabase::FieldType(const char* field_name) const
```

- `FieldSize` returns the size in bytes of the field associated with the given field index. Field indices start at 0.

```
int PgDatabase::FieldSize(int field_num) const
```

Returns the space allocated for this field in a database tuple given the field number. In other words the size of the server's binary representation of the data type. -1 is returned if the field is variable size.

- `FieldSize` returns the size in bytes of the field associated with the given field index. Field indices start at 0.

```
int PgDatabase::FieldSize(const char *field_name) const
```

Returns the space allocated for this field in a database tuple given the field name. In other words the size of the server's binary representation of the data type. -1 is returned if the field is variable size.

3.5.3. Retrieving SELECT Result Values

- `GetValue` returns a single field (column) value of one tuple of a `PGresult`. Tuple and field indices start at 0.

  ```
  const char *PgDatabase::GetValue(int tup_num, int field_num) const
  ```

  For most queries, the value returned by `GetValue` is a null-terminated string representation of the attribute value. But if `BinaryTuples` is true, the value returned by `GetValue` is the binary representation of the type in the internal format of the backend server (but not including the size word, if the field is variable-length). It is then the programmer's responsibility to cast and convert the data to the correct C type. The pointer returned by `GetValue` points to storage that is part of the `PGresult` structure. One should not modify it, and one must explicitly copy the value into other storage if it is to be used past the lifetime of the `PGresult` structure itself. `BinaryTuples` is not yet implemented.

- `GetValue` returns a single field (column) value of one tuple of a `PGresult`. Tuple and field indices start at 0.

  ```
  const char *PgDatabase::GetValue(int tup_num, const char
   *field_name) const
  ```

  For most queries, the value returned by `GetValue` is a null-terminated string representation of the attribute value. But if `BinaryTuples` is true, the value returned by `GetValue` is the binary representation of the type in the internal format of the backend server (but not including the size word, if the field is variable-length). It is then the programmer's responsibility to cast and convert the data to the correct C type. The pointer returned by `GetValue` points to storage that is part of the `PGresult` structure. One should not modify it, and one must explicitly copy the value into other storage if it is to be used past the lifetime of the `PGresult` structure itself. `BinaryTuples` is not yet implemented.

- `GetLength` returns the length of a field (column) in bytes. Tuple and field indices start at 0.

  ```
  int PgDatabase::GetLength(int tup_num, int field_num) const
  ```

  This is the actual data length for the particular data value, that is the size of the object pointed to by `GetValue`. Note that for character-represented values, this size has little to do with the binary size reported by `PQfsize`.

- `GetLength` returns the length of a field (column) in bytes. Tuple and field indices start at 0.

  ```
  int PgDatabase::GetLength(int tup_num, const char* field_name) const
  ```

  This is the actual data length for the particular data value, that is the size of the object pointed to by `GetValue`. Note that for character-represented values, this size has little to do with the binary size reported by `PQfsize`.

- `GetIsNull` returns whether a field has the null value.

  ```
  bool GetIsNull(int tup_num, int field_num) const
  ```

  Note that `GetValue` will return the empty string for null fields, not the NULL pointer.

- `GetIsNull` returns whether a field has the null value.

```
bool GetIsNull(int tup_num, const char *field_name) const
```

Note that `GetValue` will return the empty string for null fields, not the NULL pointer.

- `DisplayTuples` prints out all the tuples and, optionally, the attribute names to the specified output stream.

```
void PgDatabase::DisplayTuples(FILE *out = 0, bool fillAlign =
 true,
const char* fieldSep = "|", bool printHeader = true, bool quiet =
 false) const
```

This function is obsolescent.

- `PrintTuples` prints out all the tuples and, optionally, the attribute names to the specified output stream.

```
void PgDatabase::PrintTuples(FILE *out = 0, bool printAttName =
 true,
bool terseOutput = false, bool fillAlign = false) const
```

This function is obsolescent.

3.5.4. Retrieving Non-SELECT Result Information

- `CmdTuples` returns the number of rows affected after an `INSERT`, `UPDATE`, or `DELETE`. If the command was anything else, it returns -1.

```
int PgDatabase::CmdTuples() const
```

- `OidStatus`

```
const char *PgDatabase::OidStatus() const
```

3.6. Asynchronous Notification

PostgreSQL supports asynchronous notification via the `LISTEN` and `NOTIFY` commands. A backend registers its interest in a particular notification condition with the `LISTEN` command. All backends that are listening on a particular condition will be notified asynchronously when a `NOTIFY` of that name is executed by another backend. No additional information is passed from the notifier to the listener. Thus, typically, any actual data that needs to be communicated is transferred through a relation.

libpq++ applications are notified whenever a connected backend has received an asynchronous notification. However, the communication from the backend to the frontend is not asynchronous. The libpq++ application must poll the backend to see if there is any pending notification information. After the execution of a command, a frontend may call `PgDatabase::Notifies` to see if any notification data is currently available from the backend. `PgDatabase::Notifies` returns the notification from a list of unhandled notifications from the backend. The function returns `NULL` if there are no pending notifications from the backend. `PgDatabase::Notifies` behaves like the popping of a stack. Once a notification is returned from `PgDatabase::Notifies`, it is considered handled and will be removed from the list of notifications.

- `PgDatabase::Notifies` retrieves pending notifications from the server.

```
PGnotify* PgDatabase::Notifies()
```

The second sample program gives an example of the use of asynchronous notification.

3.7. Functions Associated with the COPY Command

The `COPY` command in PostgreSQL has options to read from or write to the network connection used by libpq++. Therefore, functions are necessary to access this network connection directly so applications may take full advantage of this capability.

- `PgDatabase::GetLine` reads a newline-terminated line of characters (transmitted by the backend server) into a buffer *`string`* of size *`length`*.

  ```
  int PgDatabase::GetLine(char* string, int length)
  ```

  Like the Unix system routine `fgets()`, this routine copies up to *`length`*`-1` characters into *`string`*. It is like `gets()`, however, in that it converts the terminating newline into a zero byte.

  `PgDatabase::GetLine` returns `EOF` at end of file, 0 if the entire line has been read, and 1 if the buffer is full but the terminating newline has not yet been read.

  Notice that the application must check to see if a new line consists of a backslash followed by a period (`\.`), which indicates that the backend server has finished sending the results of the `COPY`. Therefore, if the application ever expects to receive lines that are more than *`length`*`-1` characters long, the application must be sure to check the return value of `PgDatabase::GetLine` very carefully.

- `PgDatabase::PutLine` Sends a null-terminated *`string`* to the backend server.

  ```
  void PgDatabase::PutLine(char* string)
  ```

  The application must explicitly send the characters `\.` to indicate to the backend that it has finished sending its data.

- `PgDatabase::EndCopy` synchronizes with the backend.

  ```
  int PgDatabase::EndCopy()
  ```

  This function waits until the backend has finished processing the `COPY`. It should either be issued when the last string has been sent to the backend using `PgDatabase::PutLine` or when the last string has been received from the backend using `PgDatabase::GetLine`. It must be issued or the backend may get "out of sync" with the frontend. Upon return from this function, the backend is ready to receive the next command.

  The return value is 0 on successful completion, nonzero otherwise.

As an example:

```
PgDatabase data;
data.Exec("CREATE TABLE foo (a int4, b char(16), d double
 precision)");
data.Exec("COPY foo FROM STDIN");
data.PutLine("3\tHello World\t4.5\n");
data.PutLine("4\tGoodbye World\t7.11\n");
...
data.PutLine("\\.\n");
data.EndCopy();
```

Chapter 4. pgtcl - Tcl Binding Library

4.1. Introduction

pgtcl is a Tcl package for client programs to interface with PostgreSQL servers. It makes most of the functionality of libpq available to Tcl scripts.

This package was originally written by Jolly Chen.

Table 4.1, "`pgtcl` Commands" gives an overview over the commands available in pgtcl. These commands are described further on subsequent pages.

Table 4.1. `pgtcl` Commands

Command	Description
`pg_connect`	opens a connection to the backend server
`pg_disconnect`	closes a connection
`pg_conndefaults`	get connection options and their defaults
`pg_exec`	send a query to the backend
`pg_result`	manipulate the results of a query
`pg_select`	loop over the result of a SELECT statement
`pg_listen`	establish a callback for NOTIFY messages
`pg_lo_creat`	create a large object
`pg_lo_open`	open a large object
`pg_lo_close`	close a large object
`pg_lo_read`	read a large object
`pg_lo_write`	write a large object
`pg_lo_lseek`	seek to a position in a large object
`pg_lo_tell`	return the current seek position of a large object
`pg_lo_unlink`	delete a large object
`pg_lo_import`	import a Unix file into a large object
`pg_lo_export`	export a large object into a Unix file

The `pg_lo_*` routines are interfaces to the large object features of PostgreSQL. The functions are designed to mimic the analogous file system functions in the standard Unix file system interface. The `pg_lo_*` routines should be used within a `BEGIN/COMMIT` transaction block because the file descriptor returned by `pg_lo_open` is only valid for the current transaction. `pg_lo_import` and `pg_lo_export` *must* be used in a `BEGIN/COMMIT` transaction block.

Example 4.1, "pgtcl Example Program" shows a small example of how to use the routines.

Example 4.1. pgtcl Example Program

```
# getDBs :
#   get the names of all the databases at a given host and port number
#   with the defaults being the localhost and port 5432
```

```
#   return them in alphabetical order
proc getDBs { {host "localhost"} {port "5432"} } {
    # datnames is the list to be result
    set conn [pg_connect template1 -host $host -port $port]
    set res [pg_exec $conn "SELECT datname FROM pg_database ORDER BY
 datname"]
    set ntups [pg_result $res -numTuples]
    for {set i 0} {$i < $ntups} {incr i} {
 lappend datnames [pg_result $res -getTuple $i]
    }
    pg_result $res -clear
    pg_disconnect $conn
    return $datnames
}
```

4.2. Loading pgtcl into your application

Before using pgtcl commands, you must load `libpgtcl` into your Tcl application. This is normally done with the Tcl `load` command. Here is an example:

```
load libpgtcl[info sharedlibextension]
```

The use of `info sharedlibextension` is recommended in preference to hard-wiring `.so` or `.sl` into the program.

The `load` command will fail unless the system's dynamic loader knows where to look for the `libpgtcl` shared library file. You may need to work with `ldconfig`, or set the environment variable `LD_LIBRARY_PATH`, or use some equivalent facility for your platform to make it work. Refer to the PostgreSQL installation instructions for more information.

`libpgtcl` in turn depends on `libpq`, so the dynamic loader must also be able to find the `libpq` shared library. In practice this is seldom an issue, since both of these shared libraries are normally stored in the same directory, but it can be a stumbling block in some configurations.

If you use a custom executable for your application, you might choose to statically bind `libpgtcl` into the executable and thereby avoid the `load` command and the potential problems of dynamic linking. See the source code for pgtclsh for an example.

4.3. pgtcl Command Reference Information

Name

pg_connect — open a connection to the backend server

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_connect -conninfo connectOptions
pg_connect dbName [-host hostName]
  [-port portNumber] [-tty pqtty]
  [-options optionalBackendArgs]
```

Inputs (new style)

connectOptions

A string of connection options, each written in the form keyword = value. A list of valid options can be found in libpq's PQconnectdb() manual entry.

Inputs (old style)

dbName

Specifies a valid database name.

[-host *hostName*]

Specifies the domain name of the backend server for *dbName*.

[-port *portNumber*]

Specifies the IP port number of the backend server for *dbName*.

[-tty *pqtty*]

Specifies file or tty for optional debug output from backend.

[-options *optionalBackendArgs*]

Specifies options for the backend server for *dbName*.

Outputs

dbHandle

If successful, a handle for a database connection is returned. Handles start with the prefix pgsql.

Description

pg_connect opens a connection to the PostgreSQL backend.

Two syntaxes are available. In the older one, each possible option has a separate option switch in the pg_connect statement. In the newer form, a single option string is supplied that can contain multiple option values. See pg_conndefaults for info about the available options in the newer syntax.

Usage

Name

pg_disconnect — close a connection to the backend server

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_disconnect` *`dbHandle`*

Inputs

`dbHandle`

Specifies a valid database handle.

Outputs

None

Description

`pg_disconnect` closes a connection to the PostgreSQL backend.

Name

pg_conndefaults — obtain information about default connection parameters

Synopsis

<refsynopsisdivinfo>1998-10-07</refsynopsisdivinfo>

```
pg_conndefaults
```

Inputs

None.

Outputs

option list

The result is a list describing the possible connection options and their current default values. Each entry in the list is a sublist of the format:

```
{optname label dispchar dispsize value}
```

where the *optname* is usable as an option in `pg_connect -conninfo`.

Description

`pg_conndefaults` returns info about the connection options available in `pg_connect -conninfo` and the current default value for each option.

Usage

```
pg_conndefaults
```

Name

pg_exec — send a command string to the server

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_exec` *`dbHandle queryString`*

Inputs

`dbHandle`

Specifies a valid database handle.

`queryString`

Specifies a valid SQL query.

Outputs

`resultHandle`

A Tcl error will be returned if pgtcl was unable to obtain a backend response. Otherwise, a query result object is created and a handle for it is returned. This handle can be passed to `pg_result` to obtain the results of the query.

Description

`pg_exec` submits a query to the PostgreSQL backend and returns a result. Query result handles start with the connection handle and add a period and a result number.

Note that lack of a Tcl error is not proof that the query succeeded! An error message returned by the backend will be processed as a query result with failure status, not by generating a Tcl error in pg_exec.

Name

pg_result — get information about a query result

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_result` *`resultHandle resultOption`*

Inputs

`resultHandle`

The handle for a query result.

`resultOption`

Specifies one of several possible options.

Options

`-status`

the status of the result.

`-error`

the error message, if the status indicates error; otherwise an empty string.

`-conn`

the connection that produced the result.

`-oid`

if the command was an INSERT, the OID of the inserted tuple; otherwise an empty string.

`-numTuples`

the number of tuples returned by the query.

`-numAttrs`

the number of attributes in each tuple.

`-assign arrayName`

assign the results to an array, using subscripts of the form `(tupno,attributeName)`.

`-assignbyidx arrayName ?appendstr?`

assign the results to an array using the first attribute's value and the remaining attributes' names as keys. If *`appendstr`* is given then it is appended to each key. In short, all but the first field of each tuple are stored into the array, using subscripts of the form `(firstFieldValue,fieldNameAppendStr)`.

`-getTuple tupleNumber`

returns the fields of the indicated tuple in a list. Tuple numbers start at zero.

`-tupleArray tupleNumber arrayName`

stores the fields of the tuple in array *`arrayName`*, indexed by field names. Tuple numbers start at zero.

`-attributes`

returns a list of the names of the tuple attributes.

`-lAttributes`

returns a list of sublists, `{name ftype fsize}` for each tuple attribute.

`-clear`

clear the result query object.

Outputs

The result depends on the selected option, as described above.

Description

`pg_result` returns information about a query result created by a prior `pg_exec`.

You can keep a query result around for as long as you need it, but when you are done with it, be sure to free it by executing `pg_result -clear`. Otherwise, you have a memory leak, and Pgtcl will eventually start complaining that you've created too many query result objects.

Name

pg_select — loop over the result of a SELECT statement

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_select` *`dbHandle queryString arrayVar queryProcedure`*

Inputs

`dbHandle`

Specifies a valid database handle.

`queryString`

Specifies a valid SQL select query.

`arrayVar`

Array variable for tuples returned.

`queryProcedure`

Procedure run on each tuple found.

Outputs

None.

Description

`pg_select` submits a SELECT query to the PostgreSQL backend, and executes a given chunk of code for each tuple in the result. The *`queryString`* must be a SELECT statement. Anything else returns an error. The *`arrayVar`* variable is an array name used in the loop. For each tuple, *`arrayVar`* is filled in with the tuple field values, using the field names as the array indexes. Then the *`queryProcedure`* is executed.

In addition to the field values, the following special entries are made in the array:

`.headers`

A list of the column names returned by the SELECT.

`.numcols`

The number of columns returned by the SELECT.

`.tupno`

The current tuple number, starting at zero and incrementing for each iteration of the loop body.

Usage

This would work if table `table` has fields `control` and `name` (and, perhaps, other fields):

```
pg_select $pgconn "SELECT * FROM table" array {
 puts [format "%5d %s" $array(control) $array(name)]
}
```

Name

pg_listen — set or change a callback for asynchronous NOTIFY messages

Synopsis

<refsynopsisdivinfo>1998-5-22</refsynopsisdivinfo>

`pg_listen` *`dbHandle notifyName callbackCommand`*

Inputs

`dbHandle`

Specifies a valid database handle.

`notifyName`

Specifies the notify condition name to start or stop listening to.

`callbackCommand`

If present and not empty, provides the command string to execute when a matching notification arrives.

Outputs

None

Description

`pg_listen` creates, changes, or cancels a request to listen for asynchronous NOTIFY messages from the PostgreSQL backend. With a *`callbackCommand`* parameter, the request is established, or the command string of an already existing request is replaced. With no *`callbackCommand`* parameter, a prior request is canceled.

After a `pg_listen` request is established, the specified command string is executed whenever a NOTIFY message bearing the given name arrives from the backend. This occurs when any PostgreSQL client application issues a NOTIFY command referencing that name. (Note that the name can be, but does not have to be, that of an existing relation in the database.) The command string is executed from the Tcl idle loop. That is the normal idle state of an application written with Tk. In non-Tk Tcl shells, you can execute `update` or `vwait` to cause the idle loop to be entered.

You should not invoke the SQL statements `LISTEN` or `UNLISTEN` directly when using `pg_listen`. Pgtcl takes care of issuing those statements for you. But if you want to send a NOTIFY message yourself, invoke the SQL NOTIFY statement using `pg_exec`.

Name

pg_lo_creat — create a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_creat` *`conn mode`*

Inputs

`conn`

Specifies a valid database connection.

`mode`

Specifies the access mode for the large object

Outputs

`objOid`

The oid of the large object created.

Description

`pg_lo_creat` creates an Inversion Large Object.

Usage

mode can be any or'ing together of `INV_READ` and `INV_WRITE`. The “or” operator is `|`.

```
[pg_lo_creat $conn "INV_READ|INV_WRITE"]
```

Name

pg_lo_open — open a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_open` *`conn objOid mode`*

Inputs

`conn`

Specifies a valid database connection.

`objOid`

Specifies a valid large object oid.

`mode`

Specifies the access mode for the large object

Outputs

`fd`

A file descriptor for use in later pg_lo* routines.

Description

`pg_lo_open` open an Inversion Large Object.

Usage

Mode can be either `r`, `w`, or `rw`.

Name

pg_lo_close — close a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_close` *`conn fd`*

Inputs

`conn`

Specifies a valid database connection.

`fd`

A file descriptor for use in later pg_lo* routines.

Outputs

None

Description

`pg_lo_close` closes an Inversion Large Object.

Usage

Name

pg_lo_read — read a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_read` *`conn fd bufVar len`*

Inputs

`conn`

Specifies a valid database connection.

`fd`

File descriptor for the large object from pg_lo_open.

`bufVar`

Specifies a valid buffer variable to contain the large object segment.

`len`

Specifies the maximum allowable size of the large object segment.

Outputs

None

Description

`pg_lo_read` reads at most *`len`* bytes from a large object into a variable named *`bufVar`*.

Usage

`bufVar` must be a valid variable name.

Name

pg_lo_write — write a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_write` *`conn fd buf len`*

Inputs

`conn`

Specifies a valid database connection.

`fd`

File descriptor for the large object from pg_lo_open.

`buf`

Specifies a valid string variable to write to the large object.

`len`

Specifies the maximum size of the string to write.

Outputs

None

Description

`pg_lo_write` writes at most *`len`* bytes to a large object from a variable *`buf`*.

Usage

`buf` must be the actual string to write, not a variable name.

Name

pg_lo_lseek — seek to a position in a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_lseek` *`conn fd offset whence`*

Inputs

`conn`

Specifies a valid database connection.

`fd`

File descriptor for the large object from pg_lo_open.

`offset`

Specifies a zero-based offset in bytes.

`whence`

whence can be `SEEK_CUR`, `SEEK_END`, or `SEEK_SET`

Outputs

None

Description

`pg_lo_lseek` positions to *`offset`* bytes from the beginning of the large object.

Usage

`whence` can be `SEEK_CUR`, `SEEK_END`, or `SEEK_SET`.

Name

pg_lo_tell — return the current seek position of a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_tell` *`conn fd`*

Inputs

`conn`

Specifies a valid database connection.

`fd`

File descriptor for the large object from pg_lo_open.

Outputs

`offset`

A zero-based offset in bytes suitable for input to `pg_lo_lseek`.

Description

`pg_lo_tell` returns the current to *`offset`* in bytes from the beginning of the large object.

Usage

Name

pg_lo_unlink — delete a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_unlink` *`conn lobjId`*

Inputs

`conn`

Specifies a valid database connection.

`lobjId`

Identifier for a large object.

Outputs

None

Description

`pg_lo_unlink` deletes the specified large object.

Usage

Name

pg_lo_import — import a large object from a file

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_import` *`conn filename`*

Inputs

`conn`

Specifies a valid database connection.

`filename`

Unix file name.

Outputs

None

Description

`pg_lo_import` reads the specified file and places the contents into a large object.

Usage

`pg_lo_import` must be called within a BEGIN/END transaction block.

Name

pg_lo_export — export a large object to a file

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_lo_export conn lobjId filename
```

Inputs

conn

Specifies a valid database connection.

lobjId

Large object identifier.

filename

Unix file name.

Outputs

None

Description

`pg_lo_export` writes the specified large object into a Unix file.

Usage

`pg_lo_export` must be called within a BEGIN/END transaction block.

Chapter 5. libpgeasy - Simplified C Library

Author

Written by Bruce Momjian (<pgman@candle.pha.pa.us>) and last updated 2000-03-30

pgeasy allows you to cleanly interface to the libpq library, more like a 4GL SQL interface. Refer to Chapter 1, *libpq - C Library* for more information about libpq

It consists of set of simplified C functions that encapsulate the functionality of libpq. The functions are:

- `PGresult *doquery(char *query);`
- `PGconn *connectdb(char *options);`
- `void disconnectdb();`
- `int fetch(void *param,...);`
- `int fetchwithnulls(void *param,...);`
- `void reset_fetch();`
- `void on_error_continue();`
- `void on_error_stop();`
- `PGresult *get_result();`
- `void set_result(PGresult *newres);`
- `void unset_result(PGresult *oldres);`

Many functions return a structure or value, so you can do more work with the result if required.

You basically connect to the database with `connectdb`, issue your query with `doquery`, fetch the results with `fetch`, and finish with `disconnectdb`.

For `SELECT` queries, `fetch` allows you to pass pointers as parameters, and on return the variables are filled with data from the binary cursor you opened. These binary cursors cannot be used if you are running the pgeasy client on a system with a different architecture than the database server. If you pass a NULL pointer parameter, the column is skipped. `fetchwithnulls` allows you to retrieve the NULL status of the field by passing an `int*` after each result pointer, which returns true or false if the field is null. You can always use libpq functions on the `PGresult` pointer returned by `doquery`. `reset_fetch` starts the fetch back at the beginning.

`get_result`, `set_result`, and `unset_result` allow you to handle multiple result sets at the same time.

There are several demonstration programs in the source directory.

Chapter 6. ecpg - Embedded SQL in C

Linus Tolke
Michael Meskes

Copyright © 1996-1997 Linus Tolke
Copyright © 1998 Michael Meskes

This describes the embedded SQL package for PostgreSQL. It works with C and C++. It was written by Linus Tolke (<linus@epact.se>) and Michael Meskes (<meskes@debian.org>). The package is installed with the PostgreSQL distribution, and carries a similar license.

6.1. Why Embedded SQL?

Embedded SQL has advantages over other methods for handling SQL queries. It takes care of the tedious passing of information to and from variables in your C or C++ program. Many RDBMS packages support this embedded language.

There is an ANSI standard describing how the embedded language should work. ecpg was designed to match this standard as much as possible. It is possible to port embedded SQL programs written for other RDBMS to PostgreSQL.

6.2. The Concept

You write your program in C/C++ with special SQL constructs. When declaring variables to be used in SQL statements, you need to put them in a special `declare` section. You use a special syntax for the SQL queries.

Before compiling you run the file through the embedded SQL C preprocessor and it converts the SQL statements you used to function calls with the variables used as arguments. Both query input and result output variables are passed.

After compiling, you must link with a special library that contains needed functions. These functions fetch information from the arguments, perform the SQL query using the `libpq` interface, and put the result in the arguments specified for output.

6.3. How To Use ecpg

This section describes how to use ecpg.

6.3.1. Preprocessor

The preprocessor is called ecpg. After installation it resides in the PostgreSQL `bin/` directory.

6.3.2. Library

The ecpg library is called `libecpg.a` or `libecpg.so`. Additionally, the library uses the `libpq` library for communication to the PostgreSQL server. You will have to link your program using *`-lecpg -lpq`*.

The library has some methods that are “hidden” but may prove useful.

- `ECPGdebug(int on, FILE *stream)` turns on debug logging if called with the first argument non-zero. Debug logging is done on *`stream`*. Most SQL statement log their arguments and results.

  The most important function , `ECPGdo`, logs all SQL statements with both the expanded string, i.e. the string with all the input variables inserted, and the result from the PostgreSQL server. This can be very useful when searching for errors in your SQL statements.

- `ECPGstatus()` This method returns TRUE if we are connected to a database and FALSE if not.

6.3.3. Error handling

To detect errors from the PostgreSQL server, include a line like:

```
exec sql include sqlca;
```

in the include section of your file. This will define a struct and a variable with the name *`sqlca`* as follows:

```
struct sqlca
{
 char sqlcaid[8];
 long sqlabc;
 long sqlcode;
 struct
 {
  int sqlerrml;
  char sqlerrmc[70];
 } sqlerrm;
 char sqlerrp[8];
 long sqlerrd[6];
 /* 0: empty                                        */
 /* 1: OID of processed tuple if applicable         */
 /* 2: number of rows processed in an INSERT, UPDATE */
 /*    or DELETE statement                          */
 /* 3: empty                                        */
 /* 4: empty                                        */
 /* 5: empty                                        */
 char sqlwarn[8];
 /* 0: set to 'W' if at least one other is 'W'      */
 /* 1: if 'W' at least one character string         */
 /*    value was truncated when it was              */
 /*    stored into a host variable.                 */
 /* 2: empty                                        */
 /* 3: empty                                        */
 /* 4: empty                                        */
 /* 5: empty                                        */
 /* 6: empty                                        */
 /* 7: empty                                        */
 char sqlext[8];
} sqlca;
```

If an no error occurred in the last SQL statement. *`sqlca.sqlcode`* will be 0 (ECPG_NO_ERROR). If *`sqlca.sqlcode`* is less that zero, this is a serious error, like the database definition does not match the query. If it is greater than zero, it is a normal error like the table did not contain the requested row.

sqlca.sqlerrm.sqlerrmc will contain a string that describes the error. The string ends with the line number in the source file.

These are the errors that can occur:

-12, Out of memory in line %d.

Should not normally occur. This indicates your virtual memory is exhausted.

-200 (ECPG_UNSUPPORTED): Unsupported type %s on line %d.

Should not normally occur. This indicates the preprocessor has generated something that the library does not know about. Perhaps you are running incompatible versions of the preprocessor and the library.

-201 (ECPG_TOO_MANY_ARGUMENTS): Too many arguments line %d.

This means that PostgreSQL has returned more arguments than we have matching variables. Perhaps you have forgotten a couple of the host variables in the INTO :var1,:var2-list.

-202 (ECPG_TOO_FEW_ARGUMENTS): Too few arguments line %d.

This means that PostgreSQL has returned fewer arguments than we have host variables. Perhaps you have too many host variables in the INTO :var1,:var2-list.

-203 (ECPG_TOO_MANY_MATCHES): Too many matches line %d.

This means the query has returned several rows but the variables specified are not arrays. The SELECT command was not unique.

-204 (ECPG_INT_FORMAT): Not correctly formatted int type: %s line %d.

This means the host variable is of type int and the field in the PostgreSQL database is of another type and contains a value that cannot be interpreted as an int. The library uses strtol() for this conversion.

-205 (ECPG_UINT_FORMAT): Not correctly formatted unsigned type: %s line %d.

This means the host variable is of type unsigned int and the field in the PostgreSQL database is of another type and contains a value that cannot be interpreted as an unsigned int. The library uses strtoul() for this conversion.

-206 (ECPG_FLOAT_FORMAT): Not correctly formatted floating-point type: %s line %d.

This means the host variable is of type float and the field in the PostgreSQL database is of another type and contains a value that cannot be interpreted as a float. The library uses strtod() for this conversion.

-207 (ECPG_CONVERT_BOOL): Unable to convert %s to bool on line %d.

This means the host variable is of type bool and the field in the PostgreSQL database is neither 't' nor 'f'.

-208 (ECPG_EMPTY): Empty query line %d.

PostgreSQL returned PGRES_EMPTY_QUERY, probably because the query indeed was empty.

`-209 (ECPG_MISSING_INDICATOR): NULL value without indicator in line %d.`

PostgreSQL returned `ECPG_MISSING_INDICATOR` because a NULL was returned and no NULL indicator variable was supplied.

`-210 (ECPG_NO_ARRAY): Variable is not an array in line %d.`

PostgreSQL returned `ECPG_NO_ARRAY` because an ordinary variable was used in a place that requires an array.

`-211 (ECPG_DATA_NOT_ARRAY): Data read from backend is not an array in line %d.`

PostgreSQL returned `ECPG_DATA_NOT_ARRAY` because the database returned an ordinary variable in a place that requires array value.

`-220 (ECPG_NO_CONN): No such connection %s in line %d.`

The program tried to access a connection that does not exist.

`-221 (ECPG_NOT_CONN): Not connected in line %d.`

The program tried to access a connection that does exist but is not open.

`-230 (ECPG_INVALID_STMT): Invalid statement name %s in line %d.`

The statement you are trying to use has not been prepared.

`-240 (ECPG_UNKNOWN_DESCRIPTOR): Descriptor %s not found in line %d.`

The descriptor specified was not found. The statement you are trying to use has not been prepared.

`-241 (ECPG_INVALID_DESCRIPTOR_INDEX): Descriptor index out of range in line %d.`

The descriptor index specified was out of range.

`-242 (ECPG_UNKNOWN_DESCRIPTOR_ITEM): Descriptor %s not found in line %d.`

The descriptor specified was not found. The statement you are trying to use has not been prepared.

`-243 (ECPG_VAR_NOT_NUMERIC): Variable is not a numeric type in line %d.`

The database returned a numeric value and the variable was not numeric.

`-244 (ECPG_VAR_NOT_CHAR): Variable is not a character type in line %d.`

The database returned a non-numeric value and the variable was numeric.

`-400 (ECPG_PGSQL): Postgres error: %s line %d.`

Some PostgreSQL error. The message contains the error message from the PostgreSQL backend.

`-401 (ECPG_TRANS): Error in transaction processing line %d.`

PostgreSQL signaled that we cannot start, commit or rollback the transaction.

`-402 (ECPG_CONNECT): Could not connect to database %s in line %d.`

The connect to the database did not work.

```
100 (ECPG_NOT_FOUND): Data not found line %d.
```

This is a “normal” error that tells you that what you are querying cannot be found or you are at the end of the cursor.

6.4. Limitations

What will never be included and why it cannot be done:

Oracle's single tasking

Oracle version 7.0 on AIX 3 uses OS-supported locks in shared memory that allow an application designer to link an application in a “single tasking” way. Instead of starting one client process per application process, both the database part and the application part run in the same process. In later versions of Oracle this is no longer supported.

This would require a total redesign of the PostgreSQL access model and the performance gain does not justify the effort.

6.5. Porting From Other RDBMS Packages

The design of ecpg follows the SQL standard. Porting from a standard RDBMS should not be a problem. Unfortunately there is no such thing as a standard RDBMS. Therefore ecpg tries to understand syntax extensions as long as they do not create conflicts with the standard.

The following list shows all the known incompatibilities. If you find one not listed please notify the developers. Note, however, that we list only incompatibilities from a precompiler of another RDBMS to ecpg and not ecpg features that these RDBMS do not support.

Syntax of FETCH

The standard syntax for FETCH is:

FETCH [direction] [amount] IN|FROM *cursor*.

Oracle, however, does not use the keywords IN or FROM. This feature cannot be added since it would create parsing conflicts.

6.6. For the Developer

This section explain how ecpg works internally. It contains valuable information to help users understand how to use ecpg.

6.6.1. The Preprocessor

The first four lines written by `ecpg` to the output are fixed lines. Two are comments and two are include lines necessary to interface to the library.

Then the preprocessor reads through the file and writes output. Normally it just echoes everything to the output.

When it sees an `EXEC SQL` statement, it intervenes and changes it. The `EXEC SQL` statement can be one of these:

Declare sections

`Declare` sections begin with:

```
exec sql begin declare section;
```

and end with:

```
exec sql end declare section;
```

In this section only variable declarations are allowed. Every variable declared within this section is stored in a list of variables indexed by name together with its corresponding type.

In particular the definition of a structure or union also must be listed inside a `declare` section. Otherwise ecpg cannot handle these types since it does not know the definition.

The declaration is also echoed to the file to make it a normal C variable.

The special types `VARCHAR` and `VARCHAR2` are converted into a named struct for every variable. A declaration like:

```
VARCHAR var[180];
```

is converted into:

```
struct varchar_var { int len; char arr[180]; } var;
```

Include statements

An include statement looks like:

```
exec sql include filename;
```

Note that this is NOT the same as:

```
#include <filename.h>
```

Instead the file specified is parsed by ecpg so the contents of the file are included in the resulting C code. This way you are able to specify EXEC SQL commands in an include file.

Connect statement

A connect statement looks like:

```
exec sql connect to connection target;
```

It creates a connection to the specified database.

The *`connection target`* can be specified in the following ways:

- `dbname[@server][:port][as` *`connection name`*`][user` *`user name`*`]`
- `tcp:postgresql://server[:port][/dbname][as` *`connection name`*`][user` *`user name`*`]`
- `unix:postgresql://server[:port][/dbname][as` *`connection name`*`][user` *`user name`*`]`

- *character variable*[as *connection name*][user *user name*]
- *character string*[as *connection name*][*user*]
- default
- user

There are also different ways to specify the user name:

- *userid*
- *userid*/*password*
- *userid* identified by *password*
- *userid* using *password*

Finally, the *userid* and *password* may be a constant text, a character variable, or a character string.

Disconnect statements

A disconnect statement looks like:

```
exec sql disconnect [connection target];
```

It closes the connection to the specified database.

The *connection target* can be specified in the following ways:

- *connection name*
- default
- current
- all

Open cursor statement

An open cursor statement looks like:

```
exec sql open cursor;
```

and is not copied to the output. Instead, the cursor's DECLARE command is used because it opens the cursor as well.

Commit statement

A commit statement looks like:

```
exec sql commit;
```

Rollback statement

A rollback statement looks like:

```
exec sql rollback;
```

Other statements

Other SQL statements are used by starting with `exec sql` and ending with `;`. Everything in between is treated as an SQL statement and parsed for variable substitution.

Variable substitution occurs when a symbol starts with a colon (`:`). The variable with that name is looked up among the variables that were previously declared within a `declare` section. Depending on whether the variable is being use for input or output, a pointer to the variable is output to allow access by the function.

For every variable that is part of the SQL query, the function gets other arguments:

- The type as a special symbol.
- A pointer to the value or a pointer to the pointer.
- The size of the variable if it is a `char` or `varchar`.
- The number of elements in the array (for array fetches).
- The offset to the next element in the array (for array fetches).
- The type of the indicator variable as a special symbol.
- A pointer to the value of the indicator variable or a pointer to the pointer of the indicator variable.
- 0.
- Number of elements in the indicator array (for array fetches).
- The offset to the next element in the indicator array (for array fetches).

6.6.2. A Complete Example

Here is a complete example describing the output of the preprocessor of a file `foo.pgc`:

```
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;
```

is translated into:

```
/* 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"
```

(The indentation in this manual is added for readability and not something the preprocessor does.)

6.6.3. The Library

The most important function in the library is `ECPGdo`. It takes a variable number of arguments. Hopefully there are no computers that limit the number of variables that can be accepted by a `varargs()` function. This can easily add up to 50 or so arguments.

The arguments are:

A line number

This is a line number of the original line; used in error messages only.

A string

This is the SQL query that is to be issued. It is modified by the input variables, i.e. the variables that where not known at compile time but are to be entered in the query. Where the variables should go the string contains `?`.

Input variables

As described in the section about the preprocessor, every input variable gets ten arguments.

`ECPGt_EOIT`

An enum telling that there are no more input variables.

Output variables

As described in the section about the preprocessor, every input variable gets ten arguments. These variables are filled by the function.

ECPGt_EORT

An enum telling that there are no more variables.

In the default mode, queries are committed only when `exec sql commit` is issued. Ecpg also supports auto-commit of transactions via the `-t` command-line option or via the `exec sql set autocommit to on` statement. In `autocommit` mode, each query is automatically committed unless it is inside an explicit transaction block. This mode can be explicitly turned off using `exec sql set autocommit to off`.

Chapter 7. ODBC Interface

Tim Goeke
Thomas Lockhart

7.1. Introduction

Note

Background information originally by Tim Goeke (<tgoeke@xpressway.com>)

ODBC (Open Database Connectivity) is an abstract API that allows you to write applications that can interoperate with various RDBMS servers. ODBC provides a product-neutral interface between frontend applications and database servers, allowing a user or developer to write applications that are portable between servers from different manufacturers..

The ODBC API matches up on the backend to an ODBC-compatible data source. This could be anything from a text file to an Oracle or PostgreSQL RDBMS.

The backend access comes from ODBC drivers, or vendor-specific drivers that allow data access. psqlODBC, which is included in the PostgreSQL distribution, is such a driver, along with others that are available, such as the OpenLink ODBC drivers.

Once you write an ODBC application, you *should* be able to connect to *any* back-end database, regardless of the vendor, as long as the database schema is the same.

For example. you could have MS SQL Server and PostgreSQL servers that have exactly the same data. Using ODBC, your Windows application would make exactly the same calls and the back-end data source would look the same (to the Windows application).

7.2. Installation

In order to make use of an ODBC driver there must exist a *driver manager* on the system where the ODBC driver is to be used. There are two free ODBC driver managers for Unix-like operating systems known to us: iODBC[1] and unixODBC[2]. Instructions for installing these driver managers are to be found in the respective distribution. Software that provides database access through ODBC should provide its own driver manager (which may well be one of these two). Having said that, any driver manager that you can find for your platform should support the PostgreSQL ODBC driver, or any other ODBC driver for that matter.

Note

The unixODBC distribution ships with a PostgreSQL ODBC driver of its own, which is similar to the one contained in the PostgreSQL distribution. It is up to you which one you want to use. We plan to coordinate the development of both drivers better in the future.

To install the ODBC you simply need to supply the `--enable-odbc` option to the `configure` script when you are building the entire PostgreSQL distribution. The library will then automatically be built and

[1] http://www.iodbc.org
[2] http://www.unixodbc.org

installed with the rest of the programs. If you forget that option or want to build the ODBC driver later you can change into the directory `src/interfaces/odbc` and do `make` and `make install` there.

It is also possible to build the driver to be specifically tuned for use with iODBC or unixODBC. This means in particular that the driver will use the driver manager's routines to process the configuration files, which is probably desirable since it creates a more consistent ODBC environment on your system. If you want to do that, then supply the `configure` options `--with-iodbc` or `--with-unixodbc` (but not both).

If you build a "stand-alone" driver (not tied to iODBC or unixODBC), then you can specify where the driver should look for the configuration file `odbcinst.ini`. By default it will be the directory `/usr/local/pgsql/etc/`, or equivalent, depending on what `--prefix` and/or `--sysconfdir` options you supplied to `configure`. To select a specific location outside the PostgreSQL installation layout, use the `--with-odbcinst` option. To be most useful, it should be arranged that the driver and the driver manager read the same configuration file.

Additionally, you should install the ODBC catalog extensions. That will provide a number of functions mandated by the ODBC standard that are not supplied by PostgreSQL by default. The file `/usr/local/pgsql/share/odbc.sql` (in the default installation layout) contains the appropriate definitions, which you can install as follows:

```
psql -d template1 -f LOCATION/odbc.sql
```

where specifying `template1` as the target database will ensure that all subsequent new databases will have these same definitions. If for any reason you want to remove these functions again, run the file `odbc-drop.sql` through `psql`.

7.3. Configuration Files

`~/.odbc.ini` contains user-specified access information for the psqlODBC driver. The file uses conventions typical for Windows Registry files.

The `.odbc.ini` file has three required sections. The first is `[ODBC Data Sources]` which is a list of arbitrary names and descriptions for each database you wish to access. The second required section is the Data Source Specification and there will be one of these sections for each database. Each section must be labeled with the name given in `[ODBC Data Sources]` and must contain the following entries:

```
Driver = prefix/lib/libpsqlodbc.so
Database = DatabaseName
Servername = localhost
Port = 5432
```

Tip

Remember that the PostgreSQL database name is usually a single word, without path names of any sort. The PostgreSQL server manages the actual access to the database, and you need only specify the name from the client.

Other entries may be inserted to control the format of the display. The third required section is `[ODBC]` which must contain the `InstallDir` keyword and which may contain other options.

Here is an example `.odbc.ini` file, showing access information for three databases:

```
[ODBC Data Sources]
DataEntry = Read/Write Database
QueryOnly = Read-only Database
Test = Debugging Database
Default = Postgres Stripped
```

```
[DataEntry]
ReadOnly = 0
Servername = localhost
Database = Sales

[QueryOnly]
ReadOnly = 1
Servername = localhost
Database = Sales

[Test]
Debug = 1
CommLog = 1
ReadOnly = 0
Servername = localhost
Username = tgl
Password = "no$way"
Port = 5432
Database = test

[Default]
Servername = localhost
Database = tgl
Driver = /opt/postgres/current/lib/libpsqlodbc.so

[ODBC]
InstallDir = /opt/applix/axdata/axshlib
```

7.4. Windows Applications

In the real world, differences in drivers and the level of ODBC support lessens the potential of ODBC:

- Access, Delphi, and Visual Basic all support ODBC directly.
- Under C++, such as Visual C++, you can use the C++ ODBC API.
- In Visual C++, you can use the `CRecordSet` class, which wraps the ODBC API set within an MFC 4.2 class. This is the easiest route if you are doing Windows C++ development under Windows NT.

7.4.1. Writing Applications

“ If I write an application for PostgreSQL can I write it using ODBC calls to the PostgreSQL server, or is that only when another database program like MS SQL Server or Access needs to access the data? ”

The ODBC API is the way to go. For Visual C++ coding you can find out more at Microsoft's web site or in your Visual C++ documentation.

Visual Basic and the other RAD tools have `Recordset` objects that use ODBC directly to access data. Using the data-aware controls, you can quickly link to the ODBC back-end database (*very* quickly).

Playing around with MS Access will help you sort this out. Try using File → Get External Data.

Tip

You'll have to set up a DSN first.

7.5. ApplixWare

ApplixWare has an ODBC database interface supported on at least some platforms. ApplixWare 4.4.2 has been demonstrated under Linux with PostgreSQL 7.0 using the psqlODBC driver contained in the PostgreSQL distribution.

7.5.1. Configuration

ApplixWare must be configured correctly in order for it to be able to access the PostgreSQL ODBC software drivers.

Enabling ApplixWare Database Access

These instructions are for the 4.4.2 release of ApplixWare on Linux. Refer to the *Linux Sys Admin* on-line book for more detailed information.

1. You must modify `axnet.cnf` so that `elfodbc` can find `libodbc.so` (the ODBC driver manager) shared library. This library is included with the ApplixWare distribution, but `axnet.cnf` needs to be modified to point to the correct location.

   As root, edit the file *`applixroot`*`/applix/axdata/axnet.cnf`.

   a. At the bottom of `axnet.cnf`, find the line that starts with

      ```
      #libFor elfodbc /ax/...
      ```

   b. Change line to read

      ```
      libFor elfodbc applixroot/applix/axdata/axshlib/lib
      ```

      which will tell `elfodbc` to look in this directory for the ODBC support library. Typically Applix is installed in `/opt` so the full path would be `/opt/applix/axdata/axshlib/lib`, but if you have installed Applix somewhere else then change the path accordingly.

2. Create `.odbc.ini` as described in Section 7.3, “Configuration Files”. You may also want to add the flag

   ```
   TextAsLongVarchar=0
   ```

   to the database-specific portion of `.odbc.ini` so that text fields will not be shown as `**BLOB**`.

Testing ApplixWare ODBC Connections

1. Bring up Applix Data

2. Select the PostgreSQL database of interest.

   a. Select Query → Choose Server.

   b. Select ODBC, and click Browse. The database you configured in `.odbc.ini` should be shown. Make sure that the Host: field is empty (if it is not, `axnet` will try to contact `axnet` on another machine to look for the database).

   c. Select the database in the box that was launched by Browse, then click OK.

d. Enter user name and password in the login identification dialog, and click OK.

You should see Starting elfodbc server in the lower left corner of the data window. If you get an error dialog box, see the debugging section below.

3. The “Ready” message will appear in the lower left corner of the data window. This indicates that you can now enter queries.

4. Select a table from Query → Choose tables, and then select Query → Query to access the database. The first 50 or so rows from the table should appear.

7.5.2. Common Problems

The following messages can appear while trying to make an ODBC connection through Applix Data:

`Cannot launch gateway on server`

`elfodbc` can't find `libodbc.so`. Check your `axnet.cnf`.

`Error from ODBC Gateway: IM003::[iODBC][Driver Manager]Specified driver could not be loaded`

`libodbc.so` cannot find the driver listed in `.odbc.ini`. Verify the settings.

`Server: Broken Pipe`

The driver process has terminated due to some other problem. You might not have an up-to-date version of the PostgreSQL ODBC package.

`setuid to 256: failed to launch gateway`

The September release of ApplixWare 4.4.1 (the first release with official ODBC support under Linux) shows problems when user names exceed eight (8) characters in length. Problem description contributed by Steve Campbell (<scampbell@lear.com>).

Author

Contributed by Steve Campbell (<scampbell@lear.com>), 1998-10-20
The axnet program's security system seems a little suspect. axnet does things on behalf of the user and on a true multiuser system it really should be run with root security (so it can read/write in each user's directory). I would hesitate to recommend this, however, since we have no idea what security holes this creates.

7.5.3. Debugging ApplixWare ODBC Connections

One good tool for debugging connection problems uses the Unix system utility strace.

Debugging with `strace`

1. Start ApplixWare.

2. Start an strace on the `axnet` process. For example, if

```
$ ps -aucx | grep ax
```

shows

```
cary   10432  0.0  2.6  1740   392  ?  S  Oct  9  0:00 axnet
cary   27883  0.9 31.0 12692  4596  ?  S   10:24  0:04 axmain
```

Then run

```
$ strace -f -s 1024 -p 10432
```

3. Check the `strace` output.

Note from Cary

Many of the error messages from ApplixWare go to `stderr`, but I'm not sure where `stderr` is sent, so `strace` is the way to find out.

For example, after getting a Cannot launch gateway on server, I ran `strace` on `axnet` and got

```
[pid 27947] open("/usr/lib/libodbc.so", O_RDONLY) = -1 ENOENT (No such
 file or directory)
[pid 27947] open("/lib/libodbc.so", O_RDONLY) = -1 ENOENT (No such
 file or directory)
[pid 27947] write(2, "/usr2/applix/axdata/elfodbc: can't load library
 'libodbc.so'\n", 61) = -1 EIO (I/O error)
```

So what is happening is that `applix elfodbc` is searching for `libodbc.so`, but it cannot find it. That is why `axnet.cnf` needed to be changed.

7.5.4. Running the ApplixWare Demo

In order to go through the *ApplixWare Data Tutorial*, you need to create the sample tables that the Tutorial refers to. The ELF Macro used to create the tables tries to use a NULL condition on many of the database columns, and PostgreSQL does not currently allow this option.

To get around this problem, you can do the following:

Modifying the ApplixWare Demo

1. Copy `/opt/applix/axdata/eng/Demos/sqldemo.am` to a local directory.
2. Edit this local copy of `sqldemo.am`:
   a. Search for `null_clause = "NULL"`.
   b. Change this to `null_clause = ""`.
3. Start Applix Macro Editor.
4. Open the `sqldemo.am` file from the Macro Editor.
5. Select File → Compile and Save.
6. Exit Macro Editor.
7. Start Applix Data.
8. Select * → Run Macro.

9. Enter the value `sqldemo`, then click OK.

   You should see the progress in the status line of the data window (in the lower left corner).

10. You should now be able to access the demo tables.

7.5.5. Useful Macros

You can add information about your database login and password to the standard Applix start-up macro file. This is an example `~/axhome/macros/login.am` file:

```
macro login
set_set_system_var@("sql_username@","tgl")
set_system_var@("sql_passwd@","no$way")
endmacro
```

Caution

You should be careful about the file protections on any file containing user name and password information.

Chapter 8. JDBC Interface

Author

Originally written by Peter T. Mount (<peter@retep.org.uk>), the original author of the JDBC driver.

JDBC is a core API of Java 1.1 and later. It provides a standard set of interfaces to SQL-compliant databases.

PostgreSQL provides a *type 4* JDBC Driver. Type 4 indicates that the driver is written in Pure Java, and communicates in the database system's own network protocol. Because of this, the driver is platform independent; once compiled, the driver can be used on any system.

This chapter is not intended as a complete guide to JDBC programming, but should help to get you started. For more information refer to the standard JDBC API documentation. Also, take a look at the examples included with the source. The basic example is used here.

8.1. Setting up the JDBC Driver

8.1.1. Getting the Driver

Precompiled versions of the driver can be downloaded from the PostgreSQL JDBC web site[1].

Alternatively you can build the driver from source. Although you should only need to do this if you are making changes to the source code.

Starting with PostgreSQL version 7.1, the JDBC driver is built using Ant, a special tool for building Java-based packages. You should download Ant from the Ant web site[2] and install it before proceeding. Precompiled Ant distributions are typically set up to read a file `.antrc` in the current user's home directory for configuration. For example, to use a different JDK than the default, this may work:

```
JAVA_HOME=/usr/local/sun-jdk1.3
JAVACMD=$JAVA_HOME/bin/java
```

To build the driver, add the `--with-java` option to your `configure` command line, e.g.,

```
$ ./configure --prefix=xxx --with-java ...
```

This will build and install the driver along with the rest of the PostgreSQL package when you issue the `make/gmake` and `make/gmake install` commands. If you only want to build the driver and not the rest of PostgreSQL, change into the directory `src/interfaces/jdbc` and issue the respective `make/gmake` command there. Refer to the PostgreSQL installation instructions for more information about the configuration and build process.

When building the driver from source the jar file that is created will be named `postgresql.jar`. The build will create this file in the `src/interfaces/jdbc/jars` directory. The resulting driver will be built for the version of Java you are running. If you build with a 1.1 JDK you will build a version that supports the jdbc1 specification, if you build with a Java2 JDK (i.e. JDK1.2 or JDK1.3) you will build a version that supports the jdbc2 specification.

[1] http://jdbc.postgresql.org
[2] http://jakarta.apache.org/ant/index.html

Note

Do not try to build the driver by calling `javac` directly, as the driver uses some dynamic loading techniques for performance reasons, and `javac` cannot cope. Do not try to run `ant` directly either, because some configuration information is communicated through the makefiles. Running `ant` directly without providing these parameters will result in a broken driver.

8.1.2. Setting up the Class Path

To use the driver, the jar archive (named `postgresql.jar` if you built from source, otherwise it will likely be named `jdbc7.2-1.1.jar` or `jdbc7.2-1.2.jar` for the jdbc1 and jdbc2 versions respectively) needs to be included in the class path, either by putting it in the `CLASSPATH` environment variable, or by using flags on the `java` command line. By default, the jar archive is installed in the directory `/usr/local/pgsql/share/java`. You may have it in a different directory if you used the `--prefix` option when you ran `configure`, or if you are using a binary distribution that places it in some different location.

For instance, I have an application that uses the JDBC driver to access a large database containing astronomical objects. I have the application and the JDBC driver installed in the `/usr/local/lib` directory, and the Java JDK installed in `/usr/local/jdk1.3.1`. To run the application, I would use:

```
export CLASSPATH=/usr/local/lib/finder.jar[1]:/usr/local/pgsql/share/
java/postgresql.jar:.
java Finder
```

1 `finder.jar` contains the Finder application.

Loading the driver from within the application is covered in Section 8.2, “Using the Driver”.

8.1.3. Preparing the Database for JDBC

Because Java only uses TCP/IP connections, the PostgreSQL server must be configured to accept TCP/IP connections. This can be done by setting `tcpip_socket = true` in the `postgresql.conf` file or by supplying the `-i` option flag when starting `postmaster`.

Also, the client authentication setup in the `pg_hba.conf` file may need to be configured. Refer to the *Administrator's Guide* for details. The JDBC Driver supports trust, ident, password, md5, and crypt authentication methods.

8.2. Using the Driver

8.2.1. Importing JDBC

Any source that uses JDBC needs to import the `java.sql` package, using:

```
import java.sql.*;
```

Important

Do not import the `org.postgresql` package. If you do, your source will not compile, as `javac` will get confused.

8.2.2. Loading the Driver

Before you can connect to a database, you need to load the driver. There are two methods available, and it depends on your code which is the best one to use.

In the first method, your code implicitly loads the driver using the `Class.forName()` method. For PostgreSQL, you would use:

```
Class.forName("org.postgresql.Driver");
```

This will load the driver, and while loading, the driver will automatically register itself with JDBC.

Note

The `forName()` method can throw a `ClassNotFoundException` if the driver is not available.

This is the most common method to use, but restricts your code to use just PostgreSQL. If your code may access another database system in the future, and you do not use any PostgreSQL-specific extensions, then the second method is advisable.

The second method passes the driver as a parameter to the JVM as it starts, using the `-D` argument. Example:

```
java -Djdbc.drivers=org.postgresql.Driver example.ImageViewer
```

In this example, the JVM will attempt to load the driver as part of its initialization. Once done, the `ImageViewer` is started.

Now, this method is the better one to use because it allows your code to be used with other database packages without recompiling the code. The only thing that would also change is the connection URL, which is covered next.

One last thing: When your code then tries to open a `Connection`, and you get a No driver available `SQLException` being thrown, this is probably caused by the driver not being in the class path, or the value in the parameter not being correct.

8.2.3. Connecting to the Database

With JDBC, a database is represented by a URL (Uniform Resource Locator). With PostgreSQL, this takes one of the following forms:

- `jdbc:postgresql:`*`database`*
- `jdbc:postgresql://`*`host`*`/`*`database`*
- `jdbc:postgresql://`*`host`*`:`*`port`*`/`*`database`*

where:

`host`

The host name of the server. Defaults to `localhost`.

port

The port number the server is listening on. Defaults to the PostgreSQL standard port number (5432).

database

The database name.

To connect, you need to get a `Connection` instance from JDBC. To do this, you would use the `DriverManager.getConnection()` method:

```
Connection db = DriverManager.getConnection(url, username, password);
```

8.2.4. Closing the Connection

To close the database connection, simply call the `close()` method to the `Connection`:

```
db.close();
```

8.3. Issuing a Query and Processing the Result

Any time you want to issue SQL statements to the database, you require a `Statement` or `PreparedStatement` instance. Once you have a `Statement` or `PreparedStatement`, you can use issue a query. This will return a `ResultSet` instance, which contains the entire result. Example 8.1, "Processing a Simple Query in JDCB" illustrates this process.

Example 8.1. Processing a Simple Query in JDCB

This example will issue a simple query and print out the first column of each row using a `Statement`.

```
Statement st = db.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM mytable where columnfoo
 = 500");
while(rs.next()) {
    System.out.print("Column 1 returned ");
    System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

This example will issue the same query as before using a `PreparedStatement` and a bind value in the query.

```
int foovalue = 500;
PreparedStatement st = db.prepareStatement("SELECT * FROM mytable
 where columnfoo = ?");
st.setInt(1, foovalue);
ResultSet rs = st.executeQuery();
while(rs.next()) {
    System.out.print("Column 1 returned ");
    System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

8.3.1. Using the `Statement` or `PreparedStatement` Interface

The following must be considered when using the `Statement` or `PreparedStatement` interface:

- You can use a single `Statement` instance as many times as you want. You could create one as soon as you open the connection and use it for the connection's lifetime. But you have to remember that only one `ResultSet` can exist per `Statement` or `PreparedStatement` at a given time.
- If you need to perform a query while processing a `ResultSet`, you can simply create and use another `Statement`.
- If you are using threads, and several are using the database, you must use a separate `Statement` for each thread. Refer to Section 8.8, “Using the driver in a multi-threaded or a servlet environment” if you are thinking of using threads, as it covers some important points.
- When you are done using the `Statement` or `PreparedStatement` you should close it.

8.3.2. Using the `ResultSet` Interface

The following must be considered when using the `ResultSet` interface:

- Before reading any values, you must call `next()`. This returns true if there is a result, but more importantly, it prepares the row for processing.
- Under the JDBC specification, you should access a field only once. It is safest to stick to this rule, although at the current time, the PostgreSQL driver will allow you to access a field as many times as you want.
- You must close a `ResultSet` by calling `close()` once you have finished using it.
- Once you make another query with the `Statement` used to create a `ResultSet`, the currently open `ResultSet` instance is closed automatically.
- `ResultSet` is currently read only. You can not update data through the `ResultSet`. If you want to update data you need to do it the old fashioned way by issuing a SQL update statement. This is in conformance with the JDBC specification which does not require drivers to provide this functionality.

8.4. Performing Updates

To change data (perform an insert, update, or delete) you use the `executeUpdate()` method. `executeUpdate()` is similar to the `executeQuery()` used to issue a select, however it doesn't return a `ResultSet`, instead it returns the number of records affected by the insert, update, or delete statement.

Example 8.2. Simple Delete Example

This example will issue a simple delete and print out the number of rows deleted.

```
int foovalue = 500;
PreparedStatement st = db.prepareStatement("DELETE FROM mytable where
 columnfoo = ?");
st.setInt(1, foovalue);
int rowsDeleted = st.executeUpdate();
System.out.println(rowsDeleted + " rows deleted");
```

```
st.close();
```

8.5. Creating and Modifying Database Objects

To create, modify or drop a database object like a table or view you use the `execute()` method. `execute` is similar to the `executeQuery()` used to issue a select, however it doesn't return a result.

Example 8.3. Drop Table Example

This example will drop a table.

```
Statement st = db.createStatement();
ResultSet rs = st.executeQuery("DROP TABLE mytable");
st.close();
```

8.6. Storing Binary Data

PostgreSQL provides two distinct ways to store binary data. Binary data can be stored in a table using PostgreSQL's binary data type `bytea`, or by using the *Large Object* feature which stores the binary data in a separate table in a special format, and refers to that table by storing a value of type `OID` in your table.

In order to determine which method is appropriate you need to understand the limitations of each method. The `bytea` data type is not well suited for storing very large amounts of binary data. While a column of type `bytea` can hold upto 1Gig of binary data, it would require a huge amount of memory (RAM) to process such a large value. The Large Object method for storing binary data is better suited to storing very large values, but it has its own limitations. Specifically deleting a row that contains a Large Object does not delete the Large Object. Deleting the Large Object is a separate operation that needs to be performed. Large Objects also have some security issues since anyone connected to the database case view and/or modify any Large Object, even if they don't have permissions to view/update the row containing the Large Object.

7.2 is the first release of the JDBC Driver that supports the `bytea` data type. The introduction of this functionality in 7.2 has introduced a change in behavior as compared to previous releases. In 7.2 the methods `getBytes()`, `setBytes()`, `getBinaryStream()`, and `setBinaryStream()` operate on the `bytea` data type. In 7.1 these methods operated on the `OID` data type associated with Large Objects. It is possible to revert the driver back to the old 7.1 behavior by setting the *`compatible`* property on the `Connection` to a value of `7.1`

To use the `bytea` data type you should simply use the `getBytes()`, `setBytes()`, `getBinaryStream()`, or `setBinaryStream()` methods.

To use the Large Object functionality you can use either the `LargeObject` API provided by the PostgreSQL JDBC Driver, or by using the `getBLOB()` and `setBLOB()` methods.

Important

For PostgreSQL, you must access Large Objects within an SQL transaction. You would open a transaction by using the `setAutoCommit()` method with an input parameter of `false`.

Note

In a future release of the JDBC Driver, the `getBLOB()` and `setBLOB()` methods may no longer interact with Large Objects and will instead work on `bytea` data types. So it is recommended that you use the `LargeObject` API if you intend to use Large Objects.

Example 8.4. Binary Data Examples

For example, suppose you have a table containing the file name of an image and you also want to store the image in a `bytea` column:

```
CREATE TABLE images (imgname text, img bytea);
```

To insert an image, you would use:

```
File file = new File("myimage.gif");
FileInputStream fis = new FileInputStream(file);
PreparedStatement ps = conn.prepareStatement("INSERT INTO images
 VALUES (?, ?)");
ps.setString(1, file.getName());
ps.setBinaryStream(2, fis, file.length());
ps.executeUpdate();
ps.close();
fis.close();
```

Here, `setBinaryStream()` transfers a set number of bytes from a stream into the column of type `bytea`. This also could have been done using the `setBytes()` method if the contents of the image was already in a `byte[]`.

Retrieving an image is even easier. (We use `PreparedStatement` here, but the `Statement` class can equally be used.)

```
PreparedStatement ps = con.prepareStatement("SELECT img FROM images
 WHERE imgname=?");
ps.setString(1, "myimage.gif");
ResultSet rs = ps.executeQuery();
if (rs != null) {
    while(rs.next()) {
        byte[] imgBytes = rs.getBytes(1);
        // use the stream in some way here
    }
    rs.close();
}
ps.close();
```

Here the binary data was retrieved as an `byte[]`. You could have used a `InputStream` object instead.

Alternatively you could be storing a very large file and want to use the `LargeObject` API to store the file:

```
CREATE TABLE imagesLO (imgname text, imgOID OID);
```

To insert an image, you would use:

```
// All LargeObject API calls must be within a transaction
conn.setAutoCommit(false);

// Get the Large Object Manager to perform operations with
LargeObjectManager lobj =
 ((org.postgresql.Connection)conn).getLargeObjectAPI();

//create a new large object
int oid = lobj.create(LargeObjectManager.READ |
 LargeObjectManager.WRITE);
```

```
//open the large object for write
LargeObject obj = lobj.open(oid, LargeObjectManager.WRITE);

// Now open the file
File file = new File("myimage.gif");
FileInputStream fis = new FileInputStream(file);

// copy the data from the file to the large object
byte buf[] = new byte[2048];
int s, tl = 0;
while ((s = fis.read(buf, 0, 2048)) > 0)
{
 obj.write(buf, 0, s);
 tl += s;
}

// Close the large object
obj.close();

//Now insert the row into imagesLO
PreparedStatement ps = conn.prepareStatement("INSERT INTO imagesLO
 VALUES (?, ?)");
ps.setString(1, file.getName());
ps.setInt(2, oid);
ps.executeUpdate();
ps.close();
fis.close();
```

Retrieving the image from the Large Object:

```
// All LargeObject API calls must be within a transaction
conn.setAutoCommit(false);

// Get the Large Object Manager to perform operations with
LargeObjectManager lobj =
 ((org.postgresql.Connection)conn).getLargeObjectAPI();

PreparedStatement ps = con.prepareStatement("SELECT imgOID FROM
 imagesLO WHERE imgname=?");
ps.setString(1, "myimage.gif");
ResultSet rs = ps.executeQuery();
if (rs != null) {
    while(rs.next()) {
 //open the large object for reading
 int oid = rs.getInt(1);
 LargeObject obj = lobj.open(oid, LargeObjectManager.READ);

 //read the data
 byte buf[] = new byte[obj.size()];
 obj.read(buf, 0, obj.size());
 //do something with the data read here

 // Close the object
```

```
 obj.close();
    }
    rs.close();
}
ps.close();
```

8.7. PostgreSQL Extensions to the JDBC API

PostgreSQL is an extensible database system. You can add your own functions to the backend, which can then be called from queries, or even add your own data types. As these are facilities unique to PostgreSQL, we support them from Java, with a set of extension API's. Some features within the core of the standard driver actually use these extensions to implement Large Objects, etc.

8.7.1. Accessing the Extensions

To access some of the extensions, you need to use some extra methods in the `org.postgresql.Connection` class. In this case, you would need to case the return value of `Driver.getConnection()`. For example:

```
Connection db = Driver.getConnection(url, username, password);
// ...
// later on
Fastpath fp = ((org.postgresql.Connection)db).getFastpathAPI();
```

8.7.1.1. Class `org.postgresql.Connection`

```
public class Connection extends Object implements Connection

java.lang.Object
   |
   +----org.postgresql.Connection
```

These are the extra methods used to gain access to PostgreSQL's extensions. Methods defined by `java.sql.Connection` are not listed.

8.7.1.1.1. Methods

- `public Fastpath getFastpathAPI() throws SQLException`

  This returns the Fastpath API for the current connection. It is primarily used by the Large Object API.

  The best way to use this is as follows:

  ```
  import org.postgresql.fastpath.*;
  ...
  Fastpath fp = ((org.postgresql.Connection)myconn).getFastpathAPI();
  ```

  where `myconn` is an open `Connection` to PostgreSQL.

  Returns: Fastpath object allowing access to functions on the PostgreSQL backend.

  Throws: SQLException by Fastpath when initializing for first time

- `public LargeObjectManager getLargeObjectAPI() throws SQLException`

This returns the Large Object API for the current connection.

The best way to use this is as follows:

```
import org.postgresql.largeobject.*;
...
LargeObjectManager lo =
 ((org.postgresql.Connection)myconn).getLargeObjectAPI();
```

where `myconn` is an open `Connection` to PostgreSQL.

Returns: `LargeObject` object that implements the API

Throws: `SQLException` by `LargeObject` when initializing for first time

- `public void addDataType(String type, String name)`

  This allows client code to add a handler for one of PostgreSQL's more unique data types. Normally, a data type not known by the driver is returned by `ResultSet.getObject()` as a `PGobject` instance. This method allows you to write a class that extends `PGobject`, and tell the driver the type name, and class name to use. The down side to this, is that you must call this method each time a connection is made.

  The best way to use this is as follows:

```
 ...
((org.postgresql.Connection)myconn).addDataType("mytype","my.class.name");
 ...
```

  where `myconn` is an open `Connection` to PostgreSQL. The handling class must extend `org.postgresql.util.PGobject`.

8.7.1.2. Class `org.postgresql.Fastpath`

```
public class Fastpath extends Object

java.lang.Object
   |
   +----org.postgresql.fastpath.Fastpath
```

`Fastpath` is an API that exists within the libpq C interface, and allows a client machine to execute a function on the database backend. Most client code will not need to use this method, but it is provided because the Large Object API uses it.

To use, you need to import the `org.postgresql.fastpath` package, using the line:

```
import org.postgresql.fastpath.*;
```

Then, in your code, you need to get a `FastPath` object:

```
Fastpath fp = ((org.postgresql.Connection)conn).getFastpathAPI();
```

This will return an instance associated with the database connection that you can use to issue commands. The casing of `Connection` to `org.postgresql.Connection` is required, as the `getFastpathAPI()` is an extension method, not part of JDBC. Once you have a `Fastpath` instance, you can use the `fastpath()` methods to execute a backend function.

See Also: `FastpathFastpathArg`, `LargeObject`

8.7.1.2.1. Methods

- ```
  public Object fastpath(int fnid,
                         boolean resulttype,
                         FastpathArg args[]) throws SQLException
  ```

  Send a function call to the PostgreSQL backend.

  Parameters: *`fnid`* - Function id *`resulttype`* - True if the result is an integer, false for other results *`args`* - `FastpathArguments` to pass to fastpath

  Returns: null if no data, Integer if an integer result, or byte[] otherwise

- ```
  public Object fastpath(String name,
                         boolean resulttype,
                         FastpathArg args[]) throws SQLException
  ```

  Send a function call to the PostgreSQL backend by name.

  > **Note**
  >
  > The mapping for the procedure name to function id needs to exist, usually to an earlier call to `addfunction()`. This is the preferred method to call, as function id's can/may change between versions of the backend. For an example of how this works, refer to org.postgresql.LargeObject

  Parameters: *`name`* - Function name *`resulttype`* - True if the result is an integer, false for other results *`args`* - `FastpathArguments` to pass to fastpath

  Returns: null if no data, Integer if an integer result, or byte[] otherwise

  See Also: `LargeObject`

- ```
  public int getInteger(String name,
                        FastpathArg args[]) throws SQLException
  ```

  This convenience method assumes that the return value is an Integer

  Parameters: *`name`* - Function name *`args`* - Function arguments

  Returns: integer result

  Throws: `SQLException` if a database-access error occurs or no result

- ```
  public byte[] getData(String name,
                        FastpathArg args[]) throws SQLException
  ```

  This convenience method assumes that the return value is binary data.

  Parameters: *`name`* - Function name *`args`* - Function arguments

  Returns: byte[] array containing result

  Throws: `SQLException` if a database-access error occurs or no result

- `public void addFunction(String name, int fnid)`

  This adds a function to our look-up table. User code should use the `addFunctions` method, which is based upon a query, rather than hard coding the oid. The oid for a function is not guaranteed to remain static, even on different servers of the same version.

- `public void addFunctions(ResultSet rs) throws SQLException`

  This takes a `ResultSet` containing two columns. Column 1 contains the function name, Column 2 the oid. It reads the entire `ResultSet`, loading the values into the function table.

  ### Important

  Remember to `close()` the `ResultSet` after calling this!

  ### Implementation note about function name look-ups

  PostgreSQL stores the function id's and their corresponding names in the `pg_proc` table. To speed things up locally, instead of querying each function from that table when required, a `Hashtable` is used. Also, only the function's required are entered into this table, keeping connection times as fast as possible.

  The `org.postgresql.LargeObject` class performs a query upon its start-up, and passes the returned `ResultSet` to the `addFunctions()` method here. Once this has been done, the Large Object API refers to the functions by name.

  Do not think that manually converting them to the oid's will work. OK, they will for now, but they can change during development (there was some discussion about this for V7.0), so this is implemented to prevent any unwarranted headaches in the future.

  See Also: `LargeObjectManager`

- `public int getID(String name) throws SQLException`

  This returns the function id associated by its name If `addFunction()` or `addFunctions()` have not been called for this name, then an `SQLException` is thrown.

8.7.1.3. Class `org.postgresql.fastpath.FastpathArg`

```
public class FastpathArg extends Object

java.lang.Object
   |
   +----org.postgresql.fastpath.FastpathArg
```

Each fastpath call requires an array of arguments, the number and type dependent on the function being called. This class implements methods needed to provide this capability.

For an example on how to use this, refer to the `org.postgresql.LargeObject` package.

See Also: `Fastpath`, `LargeObjectManager`, `LargeObject`

8.7.1.3.1. Constructors

- `public FastpathArg(int value)`

Constructs an argument that consists of an integer value

Parameters: value - int value to set

- `public FastpathArg(byte bytes[])`

  Constructs an argument that consists of an array of bytes

  Parameters: bytes - array to store

- ```
  public FastpathArg(byte buf[],
                     int off,
                     int len)
  ```

  Constructs an argument that consists of part of a byte array

  Parameters:

  buf

  source array

  off

  offset within array

  len

  length of data to include

- `public FastpathArg(String s)`

  Constructs an argument that consists of a String.

8.7.2. Geometric Data Types

PostgreSQL has a set of data types that can store geometric features into a table. These include single points, lines, and polygons. We support these types in Java with the org.postgresql.geometric package. It contains classes that extend the org.postgresql.util.PGobject class. Refer to that class for details on how to implement your own data type handlers.

```
Class org.postgresql.geometric.PGbox

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGbox

   public class PGbox extends PGobject implements Serializable,
Cloneable

   This represents the box data type within PostgreSQL.

Variables
```
```

```
 public PGpoint point[]

          These are the two corner points of the box.

Constructors

 public PGbox(double x1,
              double y1,
              double x2,
              double y2)

        Parameters:
                x1 - first x coordinate
                y1 - first y coordinate
                x2 - second x coordinate
                y2 - second y coordinate

 public PGbox(PGpoint p1,
              PGpoint p2)

        Parameters:
                p1 - first point
                p2 - second point

 public PGbox(String s) throws SQLException

        Parameters:
                s - Box definition in PostgreSQL syntax

        Throws: SQLException
                if definition is invalid

 public PGbox()

          Required constructor

Methods

 public void setValue(String value) throws SQLException

          This method sets the value of this object. It should be
overridden, but still called by subclasses.

        Parameters:
                value - a string representation of the value of the
object
        Throws: SQLException
                thrown if value is invalid for this type

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)
```

```
        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overridden to allow the object to be cloned

        Overrides:
                clone in class PGobject

 public String getValue()

        Returns:
                the PGbox in the syntax expected by PostgreSQL

        Overrides:
                getValue in class PGobject

Class org.postgresql.geometric.PGcircle

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGcircle

   public class PGcircle extends PGobject implements Serializable,
Cloneable

   This represents PostgreSQL's circle data type, consisting of a
 point
and a radius

Variables

 public PGpoint center

          This is the center point

 double radius

          This is the radius

Constructors

 public PGcircle(double x,
                 double y,
```

```
                 double r)

        Parameters:
               x - coordinate of center
                y - coordinate of center
                r - radius of circle

 public PGcircle(PGpoint c,
                 double r)

        Parameters:
                c - PGpoint describing the circle's center
                r - radius of circle

 public PGcircle(String s) throws SQLException

        Parameters:
                s - definition of the circle in PostgreSQL's syntax.

        Throws: SQLException
                on conversion failure

 public PGcircle()

          This constructor is used by the driver.

Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - definition of the circle in PostgreSQL's syntax.

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two circles are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overridden to allow the object to be cloned
```

```
        Overrides:
                clone in class PGobject

 public String getValue()

        Returns:
                the PGcircle in the syntax expected by PostgreSQL

        Overrides:
                getValue in class PGobject

Class org.postgresql.geometric.PGline

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGline

   public class PGline extends PGobject implements Serializable,
Cloneable

   This implements a line consisting of two points. Currently line is
not yet implemented in the backend, but this class ensures that when
it's done were ready for it.

Variables

 public PGpoint point[]

          These are the two points.

Constructors

 public PGline(double x1,
               double y1,
               double x2,
               double y2)

        Parameters:
                x1 - coordinate for first point
                y1 - coordinate for first point
                x2 - coordinate for second point
                y2 - coordinate for second point

 public PGline(PGpoint p1,
               PGpoint p2)

        Parameters:
                p1 - first point
                p2 - second point

 public PGline(String s) throws SQLException
```

```
        Parameters:
                s - definition of the line in PostgreSQL's syntax.

        Throws: SQLException
                on conversion failure

 public PGline()

          required by the driver

Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of the line segment in PostgreSQL's
syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two lines are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overridden to allow the object to be cloned

        Overrides:
                clone in class PGobject

 public String getValue()

        Returns:
                the PGline in the syntax expected by PostgreSQL

        Overrides:
                getValue in class PGobject

Class org.postgresql.geometric.PGlseg

java.lang.Object
   |
```

```
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGlseg

   public class PGlseg extends PGobject implements Serializable,
Cloneable

   This implements a lseg (line segment) consisting of two points

Variables

 public PGpoint point[]

          These are the two points.

Constructors

 public PGlseg(double x1,
               double y1,
               double x2,
               double y2)

        Parameters:

                x1 - coordinate for first point
                y1 - coordinate for first point
                x2 - coordinate for second point
                y2 - coordinate for second point

 public PGlseg(PGpoint p1,
               PGpoint p2)

        Parameters:
                p1 - first point
                p2 - second point

 public PGlseg(String s) throws SQLException

        Parameters:
                s - Definition of the line segment in PostgreSQL's
 syntax.

        Throws: SQLException
                on conversion failure

 public PGlseg()

          required by the driver

Methods

 public void setValue(String s) throws SQLException

        Parameters:
```

```
                s - Definition of the line segment in PostgreSQL's
syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two line segments are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overridden to allow the object to be cloned

        Overrides:
               clone in class PGobject

 public String getValue()

        Returns:
                the PGlseg in the syntax expected by PostgreSQL

        Overrides:
                getValue in class PGobject

Class org.postgresql.geometric.PGpath

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGpath

   public class PGpath extends PGobject implements Serializable,
Cloneable

   This implements a path (a multiply segmented line, which may be
closed)

Variables

 public boolean open

          True if the path is open, false if closed
```

```
 public PGpoint points[]

          The points defining this path

Constructors

 public PGpath(PGpoint points[],
               boolean open)

        Parameters:
                points - the PGpoints that define the path
                open - True if the path is open, false if closed

 public PGpath()

          Required by the driver

 public PGpath(String s) throws SQLException

        Parameters:
                s - definition of the path in PostgreSQL's syntax.

        Throws: SQLException
                on conversion failure

Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of the path in PostgreSQL's syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two pathes are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overridden to allow the object to be cloned
```

```
        Overrides:
                clone in class PGobject

 public String getValue()

          This returns the path in the syntax expected by
PostgreSQL

        Overrides:
                getValue in class PGobject

 public boolean isOpen()

     This returns true if the path is open

 public boolean isClosed()

     This returns true if the path is closed

 public void closePath()

     Marks the path as closed

 public void openPath()

     Marks the path as open

Class org.postgresql.geometric.PGpoint

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGpoint

   public class PGpoint extends PGobject implements Serializable,
Cloneable

   This implements a version of java.awt.Point, except it uses double
to represent the coordinates.

   It maps to the point data type in PostgreSQL.

Variables

 public double x

          The X coordinate of the point

 public double y

          The Y coordinate of the point

Constructors
```

```
 public PGpoint(double x,
                double y)

        Parameters:
                x - coordinate
                y - coordinate

 public PGpoint(String value) throws SQLException

          This is called mainly from the other geometric types, when
 a
point is embedded within their definition.

        Parameters:
                value - Definition of this point in PostgreSQL's
syntax

 public PGpoint()

          Required by the driver

Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of this point in PostgreSQL's syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two points are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overridden to allow the object to be cloned

        Overrides:
                clone in class PGobject

 public String getValue()
```

```
        Returns:
                the PGpoint in the syntax expected by PostgreSQL

        Overrides:
                getValue in class PGobject

 public void translate(int x,
                       int y)

          Translate the point with the supplied amount.

        Parameters:
                x - integer amount to add on the x axis
                y - integer amount to add on the y axis

 public void translate(double x,
                       double y)

          Translate the point with the supplied amount.

        Parameters:
                x - double amount to add on the x axis
                y - double amount to add on the y axis

 public void move(int x,
                  int y)

          Moves the point to the supplied coordinates.

        Parameters:
                x - integer coordinate
                y - integer coordinate

public void move(double x,
                  double y)

          Moves the point to the supplied coordinates.

        Parameters:
                x - double coordinate
                y - double coordinate

 public void setLocation(int x,
                         int y)

          Moves the point to the supplied coordinates. refer to
          java.awt.Point for description of this

        Parameters:
                x - integer coordinate
                y - integer coordinate

        See Also:
```

```
                Point

 public void setLocation(Point p)

          Moves the point to the supplied java.awt.Point refer to
          java.awt.Point for description of this

        Parameters:
                p - Point to move to

        See Also:
                Point

Class org.postgresql.geometric.PGpolygon

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGpolygon

   public class PGpolygon extends PGobject implements Serializable,
Cloneable

   This implements the polygon data type within PostgreSQL.

Variables

 public PGpoint points[]

          The points defining the polygon

Constructors

 public PGpolygon(PGpoint points[])

          Creates a polygon using an array of PGpoints

        Parameters:
                points - the points defining the polygon

 public PGpolygon(String s) throws SQLException

        Parameters:
                s - definition of the polygon in PostgreSQL's syntax.

        Throws: SQLException
                on conversion failure

 public PGpolygon()

          Required by the driver

Methods
```

```
public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of the polygon in PostgreSQL's syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two polygons are identical

        Overrides:
                equals in class PGobject

public Object clone()

          This must be overridden to allow the object to be cloned

        Overrides:
                clone in class PGobject

public String getValue()

        Returns:
                the PGpolygon in the syntax expected by PostgreSQL

        Overrides:
                getValue in class PGobject
```

8.7.3. Large Objects

Large objects are supported in the standard JDBC specification. However, that interface is limited, and the API provided by PostgreSQL allows for random access to the objects contents, as if it was a local file.

The org.postgresql.largeobject package provides to Java the libpq C interface's large object API. It consists of two classes, `LargeObjectManager`, which deals with creating, opening and deleting large objects, and `LargeObject` which deals with an individual object.

8.7.3.1. Class `org.postgresql.largeobject.LargeObject`

```
public class LargeObject extends Object

java.lang.Object
   |
   +----org.postgresql.largeobject.LargeObject
```

This class implements the large object interface to PostgreSQL.

It provides the basic methods required to run the interface, plus a pair of methods that provide `InputStream` and `OutputStream` classes for this object.

Normally, client code would use the methods in `BLOB` to access large objects.

However, sometimes lower level access to Large Objects is required, that is not supported by the JDBC specification.

Refer to org.postgresql.largeobject.LargeObjectManager on how to gain access to a Large Object, or how to create one.

See Also: `LargeObjectManager`

8.7.3.1.1. Variables

public static final int SEEK_SET

Indicates a seek from the beginning of a file

public static final int SEEK_CUR

Indicates a seek from the current position

public static final int SEEK_END

Indicates a seek from the end of a file

8.7.3.1.2. Methods

- `public int getOID()`

  Returns the OID of this `LargeObject`

- `public void close() throws SQLException`

  This method closes the object. You must not call methods in this object after this is called.

- `public byte[] read(int len) throws SQLException`

  Reads some data from the object, and return as a byte[] array

- ```
  public int read(byte buf[],
                  int off,
                  int len) throws SQLException
  ```

  Reads some data from the object into an existing array

  Parameters:

  buf

  destination array

  off

  offset within array
  ```

len

number of bytes to read

- `public void write(byte buf[]) throws SQLException`

  Writes an array to the object

- ```
  public void write(byte buf[],
                    int off,
                    int len) throws SQLException
  ```

  Writes some data from an array to the object

  Parameters:

  buf

  destination array

  off

  offset within array

  len

  number of bytes to write

8.7.3.2. Class `org.postgresql.largeobject.LargeObjectManager`

```
public class LargeObjectManager extends Object

java.lang.Object
   |
   +----org.postgresql.largeobject.LargeObjectManager
```

This class implements the large object interface to PostgreSQL. It provides methods that allow client code to create, open and delete large objects from the database. When opening an object, an instance of `org.postgresql.largeobject.LargeObject` is returned, and its methods then allow access to the object.

This class can only be created by org.postgresql.Connection. To get access to this class, use the following segment of code:

```
import org.postgresql.largeobject.*;
Connection  conn;
LargeObjectManager lobj;
// ... code that opens a connection ...
lobj = ((org.postgresql.Connection)myconn).getLargeObjectAPI();
```

Normally, client code would use the `BLOB` methods to access large objects. However, sometimes lower level access to Large Objects is required, that is not supported by the JDBC specification.

Refer to org.postgresql.largeobject.LargeObject on how to manipulate the contents of a Large Object.
```

8.7.3.2.1. Variables

public static final int WRITE

This mode indicates we want to write to an object.

public static final int READ

This mode indicates we want to read an object.

public static final int READWRITE

This mode is the default. It indicates we want read and write access to a large object.

8.7.3.2.2. Methods

- `public LargeObject open(int oid) throws SQLException`

  This opens an existing large object, based on its OID. This method assumes that READ and WRITE access is required (the default).

- ```
  public LargeObject open(int oid,
                          int mode) throws SQLException
  ```

  This opens an existing large object, based on its OID, and allows setting the access mode.

- `public int create() throws SQLException`

  This creates a large object, returning its OID. It defaults to READWRITE for the new object's attributes.

- `public int create(int mode) throws SQLException`

  This creates a large object, returning its OID, and sets the access mode.

- `public void delete(int oid) throws SQLException`

  This deletes a large object.

- `public void unlink(int oid) throws SQLException`

  This deletes a large object. It is identical to the delete method, and is supplied as the C API uses “unlink”.

8.8. Using the driver in a multi-threaded or a servlet environment

A problem with many JDBC drivers is that only one thread can use a `Connection` at any one time -- otherwise a thread could send a query while another one is receiving results, and this would be a bad thing for the database engine.

The PostgreSQL JDBC Driver is thread safe. Consequently, if your application uses multiple threads then you do not have to worry about complex algorithms to ensure that only one uses the database at any time.

If a thread attempts to use the connection while another one is using it, it will wait until the other thread has finished its current operation. If it is a regular SQL statement, then the operation consists of sending the statement and retrieving any `ResultSet` (in full). If it is a `Fastpath` call (e.g., reading a block from a `LargeObject`) then it is the time to send and retrieve that block.
  ```

This is fine for applications and applets but can cause a performance problem with servlets. With servlets you can have a heavy load on the connection. If you have several threads performing queries then each but one will pause, which may not be what you are after.

To solve this, you would be advised to create a pool of connections. When ever a thread needs to use the database, it asks a manager class for a `Connection`. The manager hands a free connection to the thread and marks it as busy. If a free connection is not available, it opens one. Once the thread has finished with it, it returns it to the manager who can then either close it or add it to the pool. The manager would also check that the connection is still alive and remove it from the pool if it is dead.

So, with servlets, it is up to you to use either a single connection, or a pool. The plus side for a pool is that threads will not be hit by the bottle neck caused by a single network connection. The down side is that it increases the load on the server, as a backend process is created for each `Connection`. It is up to you and your applications requirements.

8.9. Further Reading

If you have not yet read it, I'd advise you read the JDBC API Documentation (supplied with Sun's JDK), and the JDBC Specification. Both are available from http://java.sun.com/products/jdbc/index.html.

http://jdbc.postgresql.org contains updated information not included in this document, and also includes precompiled drivers.

Chapter 9. PyGreSQL - Python Interface

Author

Written by D'Arcy J.M. Cain (<darcy@druid.net>). Based heavily on code written by Pascal Andre <andre@chimay.via.ecp.fr>. Copyright © 1995, Pascal Andre. Further modifications Copyright © 1997-2000 by D'Arcy J.M. Cain.

9.1. The pg Module

You may either choose to use the old mature interface provided by the pg module or otherwise the newer pgdb interface compliant with the DB-API 2.0[1] specification developed by the Python DB-SIG.

Here we describe only the older pg API. As long as PyGreSQL does not contain a description of the DB-API you should read about the API at http://www.python.org/topics/database/DatabaseAPI-2.0.html.

A tutorial-like introduction to the DB-API can be found at http://www2.linuxjournal.com/lj-issues/issue49/2605.html

The pg module defines three objects:

- pgobject, which handles the connection and all the requests to the database,
- pglargeobject, which handles all the accesses to PostgreSQL large objects, and
- pgqueryobject that handles query results.

If you want to see a simple example of the use of some of these functions, see http://www.druid.net/rides where I have a link at the bottom to the actual Python code for the page.

9.1.1. Constants

Some constants are defined in the pg module dictionary. They are intended to be used as a parameters for methods calls. You should refer to the libpq description (Chapter 1, *libpq - C Library*) for more information about them. These constants are:

```
INV_READ
INV_WRITE
```

large objects access modes, used by (pgobject.)locreate and (pglarge.)open.

```
SEEK_SET
SEEK_CUR
SEEK_END
```

positional flags, used by (pglarge.)seek.

```
version
__version__
```

constants that give the current version

[1] http://www.python.org/topics/database/DatabaseAPI-2.0.html

9.2. pg Module Functions

pg module defines only a few methods that allow to connect to a database and to define “default variables” that override the environment variables used by PostgreSQL.

These “default variables” were designed to allow you to handle general connection parameters without heavy code in your programs. You can prompt the user for a value, put it in the default variable, and forget it, without having to modify your environment. The support for default variables can be disabled by setting the -DNO_DEF_VAR option in the Python Setup file. Methods relative to this are specified by te tag [DV].

All variables are set to None at module initialization, specifying that standard environment variables should be used.

Name

connect — opens a connection to the database server

Synopsis

```
connect([dbname], [host], [port], [opt], [tty], [user], [passwd])
```

Parameters

dbname

Name of connected database (string/None).

host

Name of the server host (string/None).

port

Port used by the database server (integer/-1).

opt

Options for the server (string/None).

tty

File or tty for optional debug output from backend (string/None).

user

PostgreSQL user (string/None).

passwd

Password for user (string/None).

Return Type

pgobject

If successful, an object handling a database connection is returned.

Exceptions

TypeError

Bad argument type, or too many arguments.

SyntaxError

Duplicate argument definition.

pg.error

Some error occurred during pg connection definition.

(+ all exceptions relative to object allocation)

Description

This method opens a connection to a specified database on a given PostgreSQL server. You can use keywords here, as described in the Python tutorial. The names of the keywords are the name of the parameters given in the syntax line. For a precise description of the parameters, please refer to the PostgreSQL user manual.

Examples

```
import pg

con1 = pg.connect('testdb', 'myhost', 5432, None, None, 'bob', None)
con2 = pg.connect(dbname='testdb', host='localhost', user='bob')
```

Name

get_defhost — get default host name [DV]

Synopsis

```
get_defhost()
```

Parameters

none

Return Type

string or None

Default host specification

Exceptions

SyntaxError

Too many arguments.

Description

`get_defhost()` returns the current default host specification, or None if the environment variables should be used. Environment variables will not be looked up.

Name

set_defhost — set default host name [DV]

Synopsis

```
set_defhost(host)
```

Parameters

host

New default host (string/None).

Return Type

string or None

Previous default host specification.

Exceptions

TypeError

Bad argument type, or too many arguments.

Description

`set_defhost()` sets the default host value for new connections. If None is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default host.

Name

get_defport — get default port [DV]

Synopsis

```
get_defport()
```

Parameters

none

Return Type

integer or None

Default port specification

Exceptions

SyntaxError

Too many arguments.

Description

`get_defport()` returns the current default port specification, or None if the environment variables should be used. Environment variables will not be looked up.

Name

set_defport — set default port [DV]

Synopsis

```
set_defport(port)
```

Parameters

port

New default host (integer/-1).

Return Type

integer or None

Previous default port specification.

Exceptions

TypeError

Bad argument type, or too many arguments.

Description

`set_defport()` sets the default port value for new connections. If -1 is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default port.

Name

get_defopt — get default options specification [DV]

Synopsis

```
get_defopt()
```

Parameters

none

Return Type

string or None

Default options specification

Exceptions

SyntaxError

Too many arguments.

Description

`get_defopt()` returns the current default connection options specification, or None if the environment variables should be used. Environment variables will not be looked up.

Name

set_defopt — set options specification [DV]

Synopsis

```
set_defopt(options)
```

Parameters

options

New default connection options (string/None).

Return Type

string or None

Previous default opt specification.

Exceptions

TypeError

Bad argument type, or too many arguments.

Description

`set_defopt()` sets the default connection options value for new connections. If None is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default options.

Name

get_deftty — get default connection debug terminal specification [DV]

Synopsis

```
get_deftty()
```

Parameters

none

Return Type

string or None

Default debug terminal specification

Exceptions

SyntaxError

Too many arguments.

Description

`get_deftty()` returns the current default debug terminal specification, or None if the environment variables should be used. Environment variables will not be looked up.

Name

set_deftty — set default debug terminal specification [DV]

Synopsis

```
set_deftty(terminal)
```

Parameters

terminal

New default debug terminal (string/None).

Return Type

string or None

Previous default debug terminal specification.

Exceptions

TypeError

Bad argument type, or too many arguments.

Description

`set_deftty()` sets the default terminal value for new connections. If None is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default terminal.

Name

get_defbase — get default database name specification [DV]

Synopsis

```
get_defbase()
```

Parameters

none

Return Type

string or None

Default debug database name specification

Exceptions

SyntaxError

Too many arguments.

Description

`get_defbase()` returns the current default database name specification, or None if the environment variables should be used. Environment variables will not be looked up.

Name

set_defbase — set default database name specification [DV]

Synopsis

`set_defbase(`*`database`*`)`

Parameters

`database`

New default database name (string/None).

Return Type

string or None

Previous default database name specification.

Exceptions

TypeError

Bad argument type, or too many arguments.

Description

`set_defbase()` sets the default database name for new connections. If None is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default database name.

9.3. Connection object: `pgobject`

This object handles a connection to the PostgreSQL database. It embeds and hides all the parameters that define this connection, leaving just really significant parameters in function calls.

Some methods give direct access to the connection socket. They are specified by the tag [DA]. *Do not use them unless you really know what you are doing.* If you prefer disabling them, set the `-DNO_DIRECT` option in the Python `Setup` file.

Some other methods give access to large objects. if you want to forbid access to these from the module, set the `-DNO_LARGE` option in the Python `Setup` file. These methods are specified by the tag [LO].

Every `pgobject` defines a set of read-only attributes that describe the connection and its status. These attributes are:

host

the host name of the server (string)

port

the port of the server (integer)

db

the selected database (string)

options

the connection options (string)

tty

the connection debug terminal (string)

user

user name on the database system (string)

status

the status of the connection (integer: 1 - OK, 0 - BAD)

error

the last warning/error message from the server (string)

Name

query — executes a SQL command

Synopsis

`query(`*`command`*`)`

Parameters

`command`

SQL command (string).

Return Type

pgqueryobject or None

Result values.

Exceptions

TypeError

Bad argument type, or too many arguments.

ValueError

Empty SQL query.

pg.error

Error during query processing, or invalid connection.

Description

`query()` method sends a SQL query to the database. If the query is an insert statement, the return value is the OID of the newly inserted row. If it is otherwise a query that does not return a result (i.e., is not a some kind of `SELECT` statement), it returns None. Otherwise, it returns a `pgqueryobject` that can be accessed via the `getresult()` or `dictresult()` methods or simply printed.

Name

reset — resets the connection

Synopsis

```
reset()
```

Parameters

none

Return Type

none

Exceptions

TypeError

Too many (any) arguments.

Description

`reset()` method resets the current database.

Name

close — close the database connection

Synopsis

```
close()
```

Parameters

none

Return Type

none

Exceptions

TypeError

Too many (any) arguments.

Description

`close()` method closes the database connection. The connection will be closed in any case when the connection is deleted but this allows you to explicitly close it. It is mainly here to allow the DB-SIG API wrapper to implement a close function.

Name

fileno — returns the socket used to connect to the database

Synopsis

```
fileno()
```

Parameters

none

Return Type

socket id

The underlying socket id used to connect to the database.

Exceptions

TypeError

Too many (any) arguments.

Description

`fileno()` method returns the underlying socket id used to connect to the database. This is useful for use in `select` calls, etc.

Name

getnotify — gets the last notify from the server

Synopsis

```
getnotify()
```

Parameters

none

Return Type

tuple, None

Last notify from server

Exceptions

TypeError

Too many (any) arguments.

pg.error

Invalid connection.

Description

`getnotify()` method tries to get a notify from the server (from the `SQL` statement `NOTIFY`). If the server returns no notify, the methods returns None. Otherwise, it returns a tuple (couple) `(relname, pid)`, where `relname` is the name of the notify and `pid` the process id of the connection that triggered the notify. Remember to do a listen query first otherwise getnotify will always return None.

Name

inserttable — inserts a list into a table

Synopsis

`inserttable(`*`table`*`, `*`values`*`)`

Parameters

`table`

The table name (string).

`values`

The list of rows values to insert (list).

Return Type

none

Exceptions

TypeError

Bad argument type or too many (any) arguments.

pg.error

Invalid connection.

Description

`inserttable()` method allows to quickly insert large blocks of data in a table: it inserts the whole values list into the given table. The list is a list of tuples/lists that define the values for each inserted row. The rows values may contain string, integer, long or double (real) values. *Be very careful:* this method does not typecheck the fields according to the table definition; it just look whether or not it knows how to handle such types.

Name

putline — writes a line to the server socket [DA]

Synopsis

```
putline(line)
```

Parameters

line

Line to be written (string).

Return Type

none

Exceptions

TypeError

Bad argument type or too many (any) arguments.

pg.error

Invalid connection.

Description

`putline()` method allows to directly write a string to the server socket.

Name

getline — gets a line from server socket [DA]

Synopsis

```
getline()
```

Parameters

none

Return Type

string

The line read.

Exceptions

TypeError

Bad argument type or too many (any) arguments.

pg.error

Invalid connection.

Description

`getline()` method allows to directly read a string from the server socket.

Name

endcopy — synchronizes client and server [DA]

Synopsis

```
endcopy()
```

Parameters

none

Return Type

none

Exceptions

TypeError

Bad argument type or too many (any) arguments.

pg.error

Invalid connection.

Description

The use of direct access methods may desynchronize client and server. This method ensure that client and server will be synchronized.

Name

locreate — creates of large object in the database [LO]

Synopsis

`locreate(`*`mode`*`)`

Parameters

`mode`

Large object create mode.

Return Type

pglarge

Object handling the PostgreSQL large object.

Exceptions

TypeError

Bad argument type or too many arguments.

pg.error

Invalid connection, or creation error.

Description

`locreate()` method creates a large object in the database. The mode can be defined by OR-ing the constants defined in the pg module (`INV_READ and INV_WRITE`).

Name

getlo — builds a large object from given `oid` [LO]

Synopsis

`getlo(`*`oid`*`)`

Parameters

`oid`

`OID` of the existing large object (integer).

Return Type

pglarge

Object handling the PostgreSQL large object.

Exceptions

TypeError

Bad argument type or too many arguments.

pg.error

Invalid connection.

Description

`getlo()` method allows to reuse a formerly created large object through the `pglarge` interface, providing the user have its `oid`.

Name

loimport — imports a file to a PostgreSQL large object [LO]

Synopsis

`loimport(`*`filename`*`)`

Parameters

`filename`

The name of the file to be imported (string).

Return Type

pglarge

Object handling the PostgreSQL large object.

Exceptions

TypeError

Bad argument type or too many arguments.

pg.error

Invalid connection, or error during file import.

Description

`loimport()` method allows to create large objects in a very simple way. You just give the name of a file containing the data to be use.

9.4. Database wrapper class: `DB`

`pg` module contains a class called `DB`. All `pgobject` methods are included in this class also. A number of additional `DB` class methods are described below. The preferred way to use this module is as follows (See description of the initialization method below.):

```
import pg

db = pg.DB(...)

for r in db.query(
    "SELECT foo,bar
       FROM foo_bar_table
      WHERE foo !~ bar"
    ).dictresult():

    print '%(foo)s %(bar)s' % r
```

The following describes the methods and variables of this class.

The `DB` class is initialized with the same arguments as the `pg.connect` method. It also initializes a few internal variables. The statement `db = DB()` will open the local database with the name of the user just like `pg.connect()` does.

Name

pkey — returns the primary key of a table

Synopsis

```
pkey(table)
```

Parameters

table

name of table.

Return Type

string

Name of field which is the primary key of the table.

Description

`pkey()` method returns the primary key of a table. Note that this raises an exception if the table does not have a primary key.

Name

get_databases — get list of databases in the system

Synopsis

```
get_databases()
```

Parameters

none

Return Type

list

List of databases in the system.

Description

Although you can do this with a simple select, it is added here for convenience

Name

get_tables — get list of tables in connected database

Synopsis

```
get_tables()
```

Parameters

none

Return Type

list

List of tables in connected database.

Description

Although you can do this with a simple select, it is added here for convenience

Name

get_attnames — returns the attribute names of a table

Synopsis

```
get_attnames(table)
```

Parameters

table

name of table.

Return Type

dictionary

The dictionary's keys are the attribute names, the values are the type names of the attributes.

Description

Given the name of a table, digs out the set of attribute names and types.

Name

get — get a tuple from a database table

Synopsis

```
get(table, arg, [keyname])
```

Parameters

table

Name of table.

arg

Either a dictionary or the value to be looked up.

[*keyname*]

Name of field to use as key (optional).

Return Type

dictionary

A dictionary mapping attribute names to row values.

Description

This method is the basic mechanism to get a single row. It assumes that the key specifies a unique row. If keyname is not specified then the primary key for the table is used. If arg is a dictionary then the value for the key is taken from it and it is modified to include the new values, replacing existing values where necessary. The oid is also put into the dictionary but in order to allow the caller to work with multiple tables, the attribute name is munged to make it unique. It consists of the string `oid_` followed by the name of the table.

Name

insert — insert a tuple into a database table

Synopsis

```
insert(table, a)
```

Parameters

table

Name of table.

a

A dictionary of values.

Return Type

integer

The OID of the newly inserted row.

Description

This method inserts values into the table specified filling in the values from the dictionary. It then reloads the dictionary with the values from the database. This causes the dictionary to be updated with values that are modified by rules, triggers, etc.

Name

update — update a database table

Synopsis

```
update(table, a)
```

Parameters

table

Name of table.

a

A dictionary of values.

Return Type

integer

The OID of the newly updated row.

Description

Similar to insert but updates an existing row. The update is based on the OID value as munged by get. The array returned is the one sent modified to reflect any changes caused by the update due to triggers, rules, defaults, etc.

Name

clear — clear a database table

Synopsis

```
clear(table, [a])
```

Parameters

table

Name of table.

[*a*]

A dictionary of values.

Return Type

dictionary

A dictionary with an empty row.

Description

This method clears all the attributes to values determined by the types. Numeric types are set to 0, dates are set to `'today'` and everything else is set to the empty string. If the array argument is present, it is used as the array and any entries matching attribute names are cleared with everything else left unchanged.

Name

delete — deletes the row from a table

Synopsis

```
delete(table, [a])
```

Parameters

table

Name of table.

[*a*]

A dictionary of values.

Return Type

none

Description

This method deletes the row from a table. It deletes based on the OID as munged as described above.

9.5. Query result object: `pgqueryobject`

Name

getresult — gets the values returned by the query

Synopsis

```
getresult()
```

Parameters

none

Return Type

list

List of tuples.

Exceptions

SyntaxError

Too many arguments.

pg.error

Invalid previous result.

Description

`getresult()` method returns the list of the values returned by the query. More information about this result may be accessed using `listfields`, `fieldname` and `fieldnum` methods.

Name

dictresult — like getresult but returns a list of dictionaries

Synopsis

```
dictresult()
```

Parameters

none

Return Type

list

List of dictionaries.

Exceptions

SyntaxError

Too many arguments.

pg.error

Invalid previous result.

Description

`dictresult()` method returns the list of the values returned by the query with each tuple returned as a dictionary with the field names used as the dictionary index.

Name

listfields — lists the fields names of the query result

Synopsis

```
listfields()
```

Parameters

none

Return Type

list

field names

Exceptions

SyntaxError

Too many arguments.

pg.error

Invalid query result, or invalid connection.

Description

`listfields()` method returns the list of field names defined for the query result. The fields are in the same order as the result values.

Name

fieldname — field number-name conversion

Synopsis

```
fieldname(i)
```

Parameters

i

field number (integer).

Return Type

string

field name.

Exceptions

TypeError

Bad parameter type, or too many arguments.

ValueError

Invalid field number.

pg.error

Invalid query result, or invalid connection.

Description

`fieldname()` method allows to find a field name from its rank number. It can be useful for displaying a result. The fields are in the same order than the result values.

Name

fieldnum — field name-number conversion

Synopsis

`fieldnum(`*`name`*`)`

Parameters

`name`

field name (string).

Return Type

integer

field number (integer).

Exceptions

TypeError

Bad parameter type, or too many arguments.

`ValueError`

Unknown field name.

pg.error

Invalid query result, or invalid connection.

Description

`fieldnum()` method returns a field number from its name. It can be used to build a function that converts result list strings to their correct type, using a hardcoded table definition. The number returned is the field rank in the result values list.

Name

ntuples — returns the number of tuples in query object

Synopsis

```
ntuples()
```

Parameters

none

Return Type

integer

The number of tuples in query object.

Exceptions

SyntaxError

Too many arguments.

Description

`ntuples()` method returns the number of tuples found in a query.

9.6. Large Object: `pglarge`

This object handles all the request concerning a PostgreSQL large object. It embeds and hides all the "recurrent" variables (object oid and connection), exactly in the same way `pgobject`s do, thus only keeping significant parameters in function calls. It keeps a reference to the `pgobject` used for its creation, sending requests though with its parameters. Any modification but dereferencing the `pgobject` will thus affect the `pglarge` object. Dereferencing the initial `pgobject` is not a problem since Python will not deallocate it before the large object dereference it. All functions return a generic error message on call error, whatever the exact error was. The `error` attribute of the object allows to get the exact error message.

`pglarge` objects define a read-only set of attributes that allow to get some information about it. These attributes are:

oid

the oid associated with the object

pgcnx

the `pgobject` associated with the object

error

the last warning/error message of the connection

Be careful

In multithreaded environments, `error` may be modified by another thread using the same `pgobject`. Remember these object are shared, not duplicated; you should provide some locking

to be able if you want to check this. The oid attribute is very interesting because it allow you reuse the oid later, creating the `pglarge` object with a `pgobject` `getlo()` method call.

See also Chapter 2, *Large Objects* for more information about the PostgreSQL large object interface.

Name

open — opens a large object

Synopsis

`open(`*`mode`*`)`

Parameters

`mode`

open mode definition (integer).

Return Type

none

Exceptions

TypeError

Bad parameter type, or too many arguments.

IOError

Already opened object, or open error.

pg.error

Invalid connection.

Description

`open()` method opens a large object for reading/writing, in the same way than the UNIX `open()` function. The mode value can be obtained by OR-ing the constants defined in the pg module (`INV_READ, INV_WRITE`).

Name

close — closes the large object

Synopsis

```
close()
```

Parameters

none

Return Type

none

Exceptions

SyntaxError

Too many arguments.

IOError

Object is not opened, or close error.

pg.error

Invalid connection.

Description

`close()` method closes previously opened large object, in the same way than the UNIX `close()` function.

Name

read — reads from the large object

Synopsis

```
read(size)
```

Parameters

size

Maximal size of the buffer to be read (integer).

Return Type

string

The read buffer.

Exceptions

TypeError

Bad parameter type, or too many arguments.

IOError

Object is not opened, or read error.

pg.error

Invalid connection or invalid object.

Description

`read()` method allows to read data from the large object, starting at current position.

Name

write — writes to the large object

Synopsis

`write(`*`string`*`)`

Parameters

`string`

Buffer to be written (string).

Return Type

none

Exceptions

TypeError

Bad parameter type, or too many arguments.

IOError

Object is not opened, or write error.

pg.error

Invalid connection or invalid object.

Description

`write()` method allows to write data to the large object, starting at current position.

Name

seek — change current position in the large object

Synopsis

```
seek(offset, whence)
```

Parameters

offset

Position offset (integer).

whence

Positional parameter (integer).

Return Type

integer

New current position in the object.

Exceptions

TypeError

Bad parameter type, or too many arguments.

IOError

Object is not opened, or seek error.

pg.error

Invalid connection or invalid object.

Description

`seek()` method allows to move the cursor position in the large object. The whence parameter can be obtained by OR-ing the constants defined in the `pg` module (`SEEK_SET, SEEK_CUR, SEEK_END`).

Name

tell — returns current position in the large object

Synopsis

```
tell()
```

Parameters

none

Return Type

integer

Current position in the object.

Exceptions

SyntaxError

Too many arguments.

IOError

Object is not opened, or seek error.

pg.error

Invalid connection or invalid object.

Description

`tell()` method allows to get the current position in the large object.

Name

unlink — deletes the large object

Synopsis

```
unlink()
```

Parameters

none

Return Type

none

Exceptions

SyntaxError

Too many arguments.

IOError

Object is not closed, or unlink error.

pg.error

Invalid connection or invalid object.

Description

`unlink()` method unlinks (deletes) the large object.

Name

size — gives the large object size

Synopsis

```
size()
```

Parameters

none

Return Type

integer

The large object size.

Exceptions

SyntaxError

Too many arguments.

IOError

Object is not opened, or seek/tell error.

pg.error

Invalid connection or invalid object.

Description

`size()` method allows to get the size of the large object. It was implemented because this function is very useful for a WWW interfaced database. Currently the large object needs to be opened.

Name

export — saves the large object to file

Synopsis

`export(`*`filename`*`)`

Parameters

`filename`

The file to be created.

Return Type

none

Exceptions

TypeError

Bad argument type, or too many arguments.

IOError

Object is not closed, or export error.

pg.error

Invalid connection or invalid object.

Description

`export()` method allows to dump the content of a large object in a very simple way. The exported file is created on the host of the program, not the server host.

9.7. DB-API Interface

See http://www.python.org/topics/database/DatabaseAPI-2.0.html for a description of the DB-API 2.0.

Part II. Server Programming

This second part of the manual explains the PostgreSQL approach to extensibility and describe how users can extend PostgreSQL by adding user-defined types, operators, aggregates, and both query language and programming language functions. After a discussion of the PostgreSQL rule system, we discuss the trigger and SPI interfaces.

Table of Contents

10. Architecture 174
    10.1. PostgreSQL Architectural Concepts 174
11. Extending SQL: An Overview 177
    11.1. How Extensibility Works 177
    11.2. The PostgreSQL Type System 177
    11.3. About the PostgreSQL System Catalogs 177
12. Extending SQL: Functions 181
    12.1. Introduction 181
    12.2. Query Language (SQL) Functions 181
        12.2.1. Examples 181
        12.2.2. SQL Functions on Base Types 182
        12.2.3. SQL Functions on Composite Types 183
        12.2.4. SQL Functions Returning Sets 184
    12.3. Procedural Language Functions 185
    12.4. Internal Functions 185
    12.5. C Language Functions 186
        12.5.1. Dynamic Loading 186
        12.5.2. Base Types in C-Language Functions 187
        12.5.3. Version-0 Calling Conventions for C-Language Functions 190
        12.5.4. Version-1 Calling Conventions for C-Language Functions 192
        12.5.5. Composite Types in C-Language Functions 194
        12.5.6. Writing Code 195
        12.5.7. Compiling and Linking Dynamically-Loaded Functions 196
    12.6. Function Overloading 198
    12.7. Procedural Language Handlers 199
13. Extending SQL: Types 202
14. Extending SQL: Operators 204
    14.1. Introduction 204
    14.2. Example 204
    14.3. Operator Optimization Information 205
        14.3.1. COMMUTATOR 205
        14.3.2. NEGATOR 205
        14.3.3. RESTRICT 206
        14.3.4. JOIN 207
        14.3.5. HASHES 207
        14.3.6. SORT1 and SORT2 208
15. Extending SQL: Aggregates 209
16. The Rule System 211
    16.1. Introduction 211
    16.2. What is a Query Tree? 211
        16.2.1. The Parts of a Query tree 211
    16.3. Views and the Rule System 213
        16.3.1. Implementation of Views in PostgreSQL 213
        16.3.2. How SELECT Rules Work 213
        16.3.3. View Rules in Non-SELECT Statements 218
        16.3.4. The Power of Views in PostgreSQL 219
        16.3.5. What about updating a view? 220
    16.4. Rules on INSERT, UPDATE and DELETE 220
        16.4.1. Differences from View Rules 220
        16.4.2. How These Rules Work 220
        16.4.3. Cooperation with Views 224

16.5. Rules and Permissions 230
16.6. Rules versus Triggers 231
17. Interfacing Extensions To Indexes 234
17.1. Introduction 234
17.2. Access Methods 234
17.3. Access Method Strategies 235
17.4. Access Method Support Routines 235
17.5. Operator Classes 236
17.6. Creating the Operators and Support Routines 237
18. Index Cost Estimation Functions 240
19. GiST Indexes 243
20. Triggers 245
20.1. Trigger Creation 245
20.2. Interaction with the Trigger Manager 246
20.3. Visibility of Data Changes 248
20.4. Examples 249
21. Server Programming Interface 252
21.1. Interface Functions 252
21.2. Interface Support Functions 266
21.3. Memory Management 273
21.4. Visibility of Data Changes 285
21.5. Examples 285

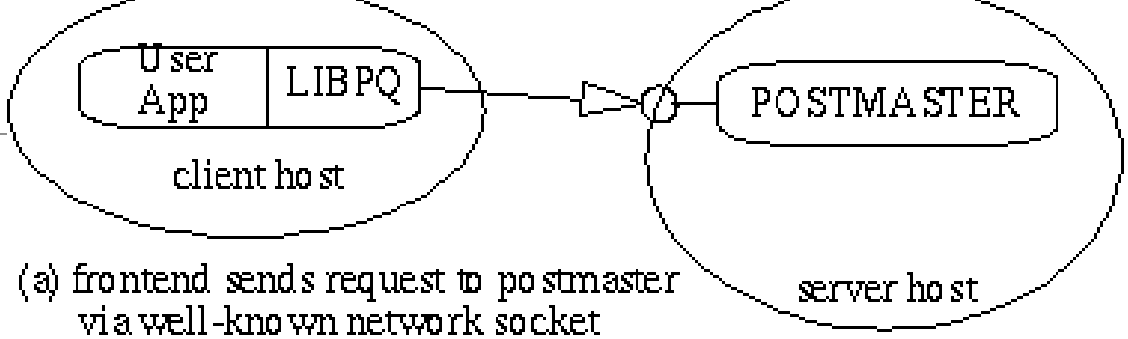
Chapter 10. Architecture

10.1. PostgreSQL Architectural Concepts

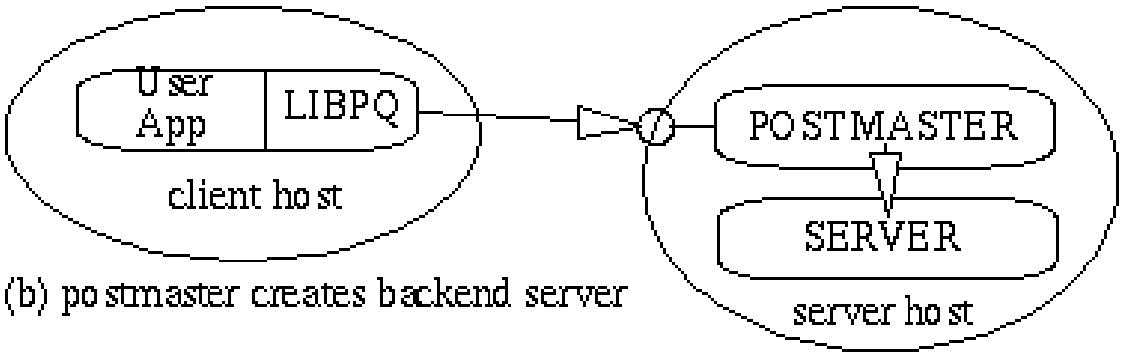
Before we begin, you should understand the basic PostgreSQL system architecture. Understanding how the parts of PostgreSQL interact will make the next chapter somewhat clearer. In database jargon, PostgreSQL uses a simple "process per-user" client/server model. A PostgreSQL session consists of the following cooperating Unix processes (programs):

- A supervisory daemon process (the postmaster),
- the user's frontend application (e.g., the psql program), and
- one or more backend database servers (the postgres process itself).

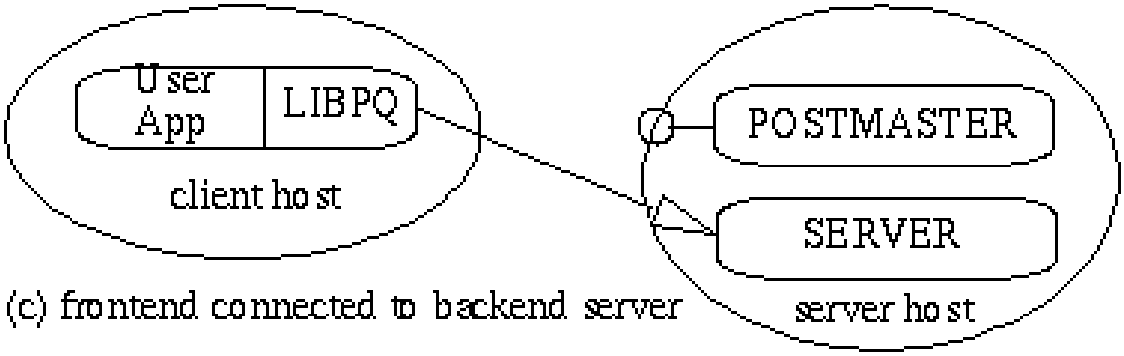
A single postmaster manages a given collection of databases on a single host. Such a collection of databases is called a cluster (of databases). A frontend application that wishes to access a given database within a cluster makes calls to an interface library (e.g., libpq) that is linked into the application. The library sends user requests over the network to the postmaster (Figure 10.1, “How a connection is established”(a)), which in turn starts a new backend server process (Figure 10.1, “How a connection is established”(b))



(a) frontend sends request to postmaster via well-known network socket

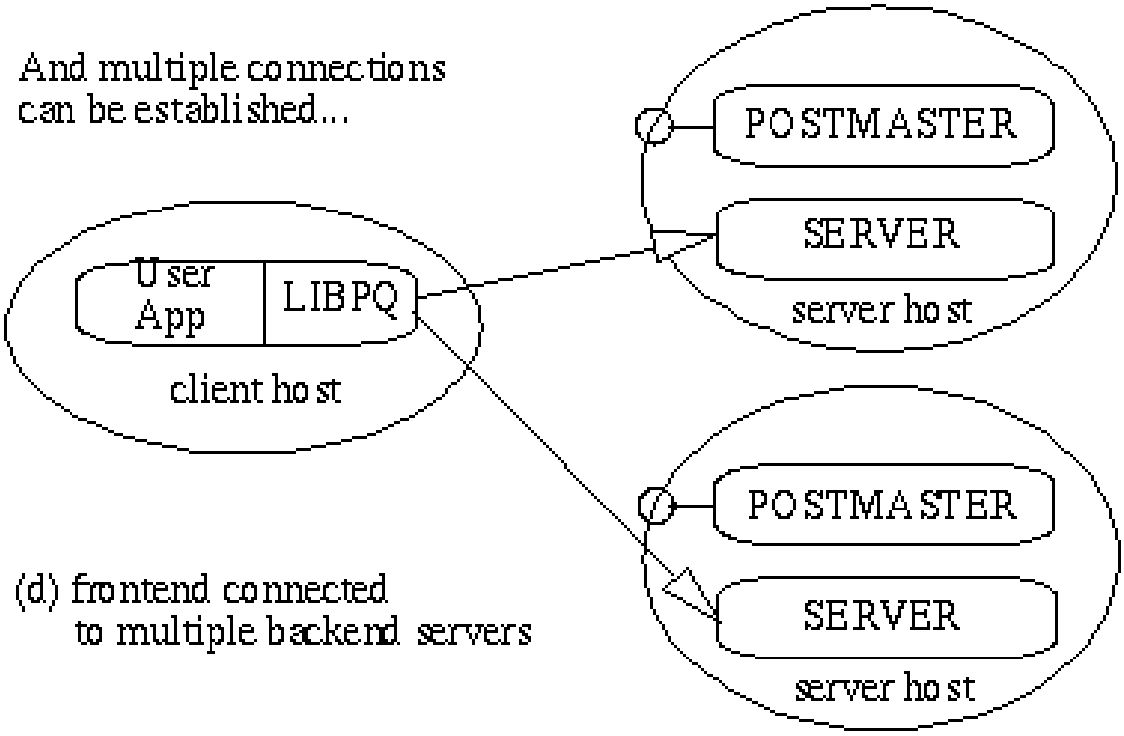


(b) postmaster creates backend server



(c) frontend connected to backend server

And multiple connections can be established...



(d) frontend connected to multiple backend servers

and connects the frontend process to the new server (Figure 10.1, “How a connection is established”(c)). From that point on, the frontend process and the backend server communicate without intervention by the postmaster. Hence, the postmaster is always running, waiting for connection requests, whereas frontend and backend processes come and go. The `libpq` library allows a single frontend to make multiple connections to backend processes. However, each backend process is a single-threaded process that can only execute one query at a time; so the communication over any one frontend-to-backend connection is single-threaded.

One implication of this architecture is that the postmaster and the backend always run on the same machine (the database server), while the frontend application may run anywhere. You should keep this in mind, because the files that can be accessed on a client machine may not be accessible (or may only be accessed using a different path name) on the database server machine.

You should also be aware that the postmaster and postgres servers run with the user ID of the PostgreSQL “superuser”. Note that the PostgreSQL superuser does not have to be any particular user (e.g., a user named `postgres`), although many systems are installed that way. Furthermore, the PostgreSQL superuser should definitely not be the Unix superuser, `root`! It is safest if the PostgreSQL superuser is an ordinary, unprivileged user so far as the surrounding Unix system is concerned. In any case, all files relating to a database should belong to this Postgres superuser.

Chapter 11. Extending SQL: An Overview

In the sections that follow, we will discuss how you can extend the PostgreSQL SQL query language by adding:

- functions
- data types
- operators
- aggregates

11.1. How Extensibility Works

PostgreSQL is extensible because its operation is catalog-driven. If you are familiar with standard relational systems, you know that they store information about databases, tables, columns, etc., in what are commonly known as system catalogs. (Some systems call this the data dictionary). The catalogs appear to the user as tables like any other, but the DBMS stores its internal bookkeeping in them. One key difference between PostgreSQL and standard relational systems is that PostgreSQL stores much more information in its catalogs -- not only information about tables and columns, but also information about its types, functions, access methods, and so on. These tables can be modified by the user, and since PostgreSQL bases its internal operation on these tables, this means that PostgreSQL can be extended by users. By comparison, conventional database systems can only be extended by changing hardcoded procedures within the DBMS or by loading modules specially written by the DBMS vendor.

PostgreSQL is also unlike most other data managers in that the server can incorporate user-written code into itself through dynamic loading. That is, the user can specify an object code file (e.g., a shared library) that implements a new type or function and PostgreSQL will load it as required. Code written in SQL is even more trivial to add to the server. This ability to modify its operation “on the fly” makes PostgreSQL uniquely suited for rapid prototyping of new applications and storage structures.

11.2. The PostgreSQL Type System

The PostgreSQL type system can be broken down in several ways. Types are divided into base types and composite types. Base types are those, like `int4`, that are implemented in a language such as C. They generally correspond to what are often known as *abstract data types*; PostgreSQL can only operate on such types through methods provided by the user and only understands the behavior of such types to the extent that the user describes them. Composite types are created whenever the user creates a table.

PostgreSQL stores these types in only one way (within the file that stores all rows of a table) but the user can “look inside” at the attributes of these types from the query language and optimize their retrieval by (for example) defining indexes on the attributes. PostgreSQL base types are further divided into built-in types and user-defined types. Built-in types (like `int4`) are those that are compiled into the system. User-defined types are those created by the user in the manner to be described later.

11.3. About the PostgreSQL System Catalogs

Having introduced the basic extensibility concepts, we can now take a look at how the catalogs are actually laid out. You can skip this section for now, but some later sections will be incomprehensible without the

information given here, so mark this page for later reference. All system catalogs have names that begin with `pg_`. The following tables contain information that may be useful to the end user. (There are many other system catalogs, but there should rarely be a reason to query them directly.)

Table 11.1. PostgreSQL System Catalogs

Catalog Name	Description
pg_database	databases
pg_class	tables
pg_attribute	table columns
pg_index	indexes
pg_proc	procedures/functions
pg_type	data types (both base and complex)
pg_operator	operators
pg_aggregate	aggregate functions
pg_am	access methods
pg_amop	access method operators
pg_amproc	access method support functions
pg_opclass	access method operator classes

Figure 11.1. The major PostgreSQL system catalogs

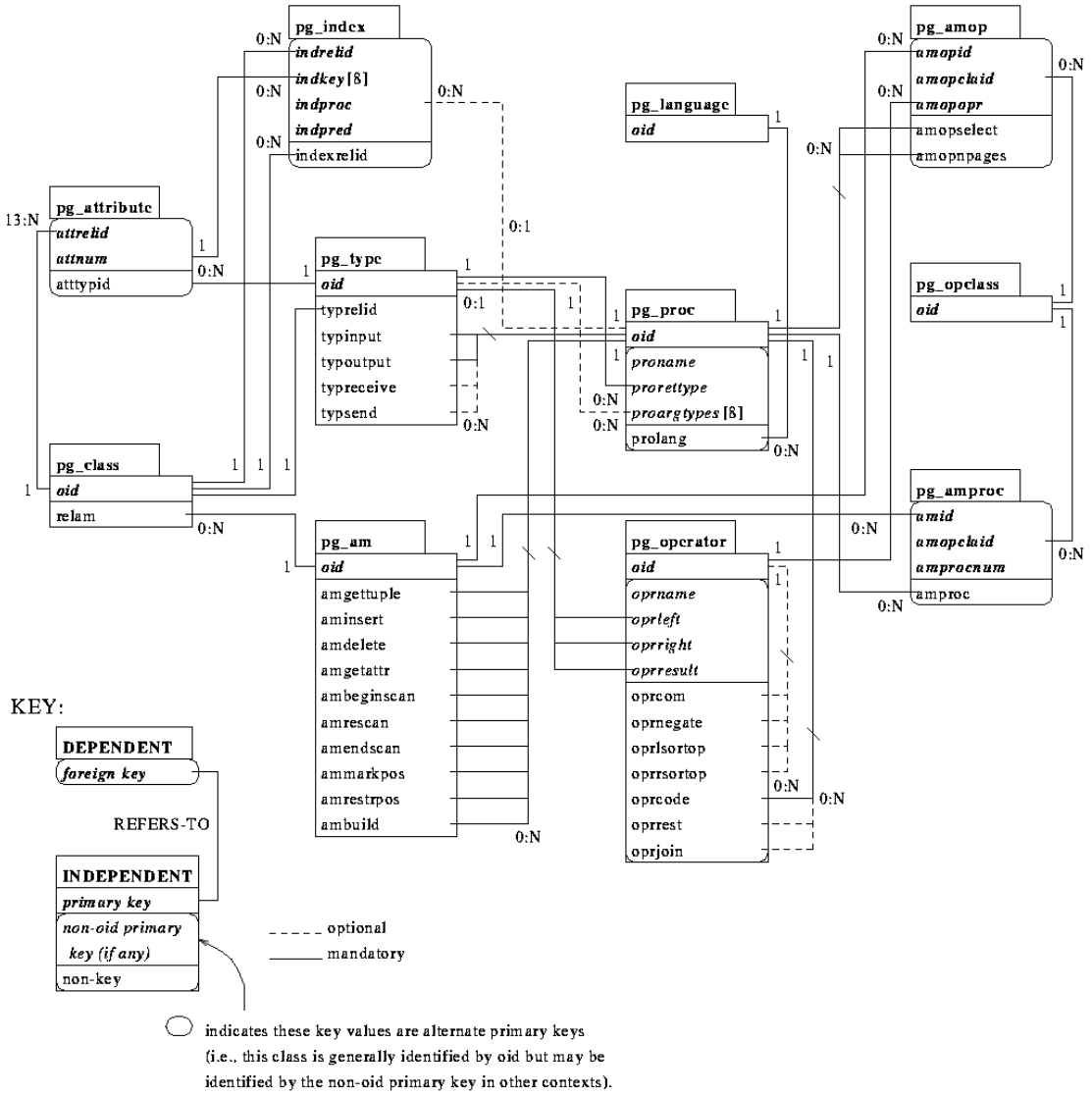


Figure 3. The major POSTGRES system catalogs.

The *Developer's Guide* gives a more detailed explanation of these catalogs and their columns. However, Figure 11.1, “The major PostgreSQL system catalogs” shows the major entities and their relationships in the system catalogs. (Columns that do not refer to other entities are not shown unless they are part of a primary key.) This diagram is more or less incomprehensible until you actually start looking at the contents of the catalogs and see how they relate to each other. For now, the main things to take away from this diagram are as follows:

- In several of the sections that follow, we will present various join queries on the system catalogs that display information we need to extend the system. Looking at this diagram should make some of these join queries (which are often three- or four-way joins) more understandable, because you will be able to see that the columns used in the queries form foreign keys in other tables.
- Many different features (tables, columns, functions, types, access methods, etc.) are tightly integrated in this schema. A simple create command may modify many of these catalogs.
- Types and procedures are central to the schema.

Note

We use the words *procedure* and *function* more or less interchangeably.

Nearly every catalog contains some reference to rows in one or both of these tables. For example, PostgreSQL frequently uses type signatures (e.g., of functions and operators) to identify unique rows of other catalogs.

- There are many columns and relationships that have obvious meanings, but there are many (particularly those that have to do with access methods) that do not. The relationships between `pg_am`, `pg_amop`, `pg_amproc`, `pg_operator`, and `pg_opclass` are particularly hard to understand and will be described in depth (in Chapter 17, *Interfacing Extensions To Indexes*) after we have discussed basic extensions.

Chapter 12. Extending SQL: Functions

12.1. Introduction

As it turns out, part of defining a new type is the definition of functions that describe its behavior. Consequently, while it is possible to define a new function without defining a new type, the reverse is not true. We therefore describe how to add new functions to PostgreSQL before describing how to add new types.

PostgreSQL provides four kinds of functions:

- query language functions (functions written in SQL)
- procedural language functions (functions written in, for example, PL/Tcl or PL/pgSQL)
- internal functions
- C language functions

Every kind of function can take a base type, a composite type, or some combination as arguments (parameters). In addition, every kind of function can return a base type or a composite type. It's easiest to define SQL functions, so we'll start with those. Examples in this section can also be found in `funcs.sql` and `funcs.c` in the tutorial directory.

Throughout this chapter, it can be useful to look at the reference page of the `CREATE FUNCTION` command to understand the examples better.

12.2. Query Language (SQL) Functions

SQL functions execute an arbitrary list of SQL statements, returning the result of the last query in the list, which must be a `SELECT`. In the simple (non-set) case, the first row of the last query's result will be returned. (Bear in mind that "the first row" of a multi-row result is not well-defined unless you use `ORDER BY`.) If the last query happens to return no rows at all, NULL will be returned.

Alternatively, an SQL function may be declared to return a set, by specifying the function's return type as `SETOF` *`sometype`*. In this case all rows of the last query's result are returned. Further details appear below.

The body of an SQL function should be a list of one or more SQL statements separated by semicolons. Note that because the syntax of the `CREATE FUNCTION` command requires the body of the function to be enclosed in single quotes, single quote marks (`'`) used in the body of the function must be escaped, by writing two single quotes (`''`) or a backslash (`\'`) where each quote is desired.

Arguments to the SQL function may be referenced in the function body using the syntax `$`*`n`*: $1 refers to the first argument, $2 to the second, and so on. If an argument is of a composite type, then the "dot notation", e.g., `$1.emp`, may be used to access attributes of the argument.

12.2.1. Examples

To illustrate a simple SQL function, consider the following, which might be used to debit a bank account:

```
CREATE FUNCTION tp1 (integer, numeric) RETURNS integer AS '
    UPDATE bank
        SET balance = balance - $2
```

```
        WHERE accountno = $1;
    SELECT 1;
' LANGUAGE SQL;
```

A user could execute this function to debit account 17 by $100.00 as follows:

```
SELECT tp1(17, 100.0);
```

In practice one would probably like a more useful result from the function than a constant "1", so a more likely definition is

```
CREATE FUNCTION tp1 (integer, numeric) RETURNS numeric AS '
    UPDATE bank
        SET balance = balance - $2
        WHERE accountno = $1;
    SELECT balance FROM bank WHERE accountno = $1;
' LANGUAGE SQL;
```

which adjusts the balance and returns the new balance.

Any collection of commands in the SQL language can be packaged together and defined as a function. The commands can include data modification (i.e., `INSERT`, `UPDATE`, and `DELETE`) as well as `SELECT` queries. However, the final command must be a `SELECT` that returns whatever is specified as the function's return type.

```
CREATE FUNCTION clean_EMP () RETURNS integer AS '
    DELETE FROM EMP
        WHERE EMP.salary <= 0;
    SELECT 1 AS ignore_this;
' LANGUAGE SQL;

SELECT clean_EMP();

 x
---
 1
```

12.2.2. SQL Functions on Base Types

The simplest possible SQL function has no arguments and simply returns a base type, such as `integer`:

```
CREATE FUNCTION one() RETURNS integer AS '
    SELECT 1 as RESULT;
' LANGUAGE SQL;

SELECT one();

 one
-----
   1
```

Notice that we defined a column alias within the function body for the result of the function (with the name `RESULT`), but this column alias is not visible outside the function. Hence, the result is labeled `one` instead of `RESULT`.

It is almost as easy to define SQL functions that take base types as arguments. In the example below, notice how we refer to the arguments within the function as `$1` and `$2`:

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS '
    SELECT $1 + $2;
' LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

12.2.3. SQL Functions on Composite Types

When specifying functions with arguments of composite types, we must not only specify which argument we want (as we did above with `$1` and `$2`) but also the attributes of that argument. For example, suppose that `EMP` is a table containing employee data, and therefore also the name of the composite type of each row of the table. Here is a function `double_salary` that computes what your salary would be if it were doubled:

```
CREATE FUNCTION double_salary(EMP) RETURNS integer AS '
    SELECT $1.salary * 2 AS salary;
' LANGUAGE SQL;

SELECT name, double_salary(EMP) AS dream
    FROM EMP
    WHERE EMP.cubicle ~= point '(2,1)';

 name | dream
------+-------
 Sam  |  2400
```

Notice the use of the syntax `$1.salary` to select one field of the argument row value. Also notice how the calling `SELECT` command uses a table name to denote the entire current row of that table as a composite value.

It is also possible to build a function that returns a composite type. (However, as we'll see below, there are some unfortunate restrictions on how the function may be used.) This is an example of a function that returns a single `EMP` row:

```
CREATE FUNCTION new_emp() RETURNS EMP AS '
    SELECT text ''None'' AS name,
        1000 AS salary,
        25 AS age,
        point ''(2,2)'' AS cubicle;
' LANGUAGE SQL;
```

In this case we have specified each of the attributes with a constant value, but any computation or expression could have been substituted for these constants. Note two important things about defining the function:

- The target list order must be exactly the same as that in which the columns appear in the table associated with the composite type.
- You must typecast the expressions to match the definition of the composite type, or you will get errors like this:

```
 ERROR:  function declared to return emp returns varchar instead of
  text at column 1
```

In the present release of PostgreSQL there are some unpleasant restrictions on how functions returning composite types can be used. Briefly, when calling a function that returns a row, we cannot retrieve the entire row. We must either project a single attribute out of the row or pass the entire row into another function. (Trying to display the entire row value will yield a meaningless number.) For example,

```
SELECT name(new_emp());

 name
------
 None
```

This example makes use of the function notation for projecting attributes. The simple way to explain this is that we can usually use the notations `attribute(table)` and `table.attribute` interchangeably:

```
--
-- this is the same as:
--  SELECT EMP.name AS youngster FROM EMP WHERE EMP.age < 30
--
SELECT name(EMP) AS youngster
    FROM EMP
    WHERE age(EMP) < 30;

 youngster
-----------
 Sam
```

The reason why, in general, we must use the function syntax for projecting attributes of function return values is that the parser just doesn't understand the dot syntax for projection when combined with function calls.

```
SELECT new_emp().name AS nobody;
ERROR:  parser: parse error at or near "."
```

Another way to use a function returning a row result is to declare a second function accepting a row type parameter, and pass the function result to it:

```
CREATE FUNCTION getname(emp) RETURNS text AS
'SELECT $1.name;'
LANGUAGE SQL;

SELECT getname(new_emp());
 getname
---------
 None
(1 row)
```

12.2.4. SQL Functions Returning Sets

As previously mentioned, an SQL function may be declared as returning `SETOF` *`sometype`*. In this case the function's final `SELECT` query is executed to completion, and each row it outputs is returned as an element of the set.

Functions returning sets may only be called in the target list of a `SELECT` query. For each row that the `SELECT` generates by itself, the function returning set is invoked, and an output row is generated for each element of the function's result set. An example:

```
CREATE FUNCTION listchildren(text) RETURNS SETOF text AS
'SELECT name FROM nodes WHERE parent = $1'
LANGUAGE SQL;

SELECT * FROM nodes;
   name    | parent
-----------+--------
 Top       |
 Child1    | Top
 Child2    | Top
 Child3    | Top
 SubChild1 | Child1
 SubChild2 | Child1
(6 rows)

SELECT listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, listchildren(name) FROM nodes;
  name   | listchildren
--------+--------------
 Top     | Child1
 Top     | Child2
 Top     | Child3
 Child1  | SubChild1
 Child1  | SubChild2
(5 rows)
```

In the last `SELECT`, notice that no output row appears for `Child2`, `Child3`, etc. This happens because `listchildren` returns an empty set for those inputs, so no output rows are generated.

12.3. Procedural Language Functions

Procedural languages aren't built into the PostgreSQL server; they are offered by loadable modules. Please refer to the documentation of the procedural language in question for details about the syntax and how the function body is interpreted for each language.

There are currently four procedural languages available in the standard PostgreSQL distribution: PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python. Other languages can be defined by users. Refer to Chapter 22, *Procedural Languages* for more information. The basics of developing a new procedural language are covered in Section 12.7, “Procedural Language Handlers”.

12.4. Internal Functions

Internal functions are functions written in C that have been statically linked into the PostgreSQL server. The “body” of the function definition specifies the C-language name of the function, which need not be the same as the name being declared for SQL use. (For reasons of backwards compatibility, an empty body is accepted as meaning that the C-language function name is the same as the SQL name.)

Normally, all internal functions present in the backend are declared during the initialization of the database cluster (`initdb`), but a user could use `CREATE FUNCTION` to create additional alias names for an internal function. Internal functions are declared in `CREATE FUNCTION` with language name `internal`. For instance, to create an alias for the `sqrt` function:

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
    AS 'dsqrt'
    LANGUAGE INTERNAL
    WITH (isStrict);
```

(Most internal functions expect to be declared “strict”.)

Note

Not all “predefined” functions are “internal” in the above sense. Some predefined functions are written in SQL.

12.5. C Language Functions

User-defined functions can be written in C (or a language that can be made compatible with C, such as C ++). Such functions are compiled into dynamically loadable objects (also called shared libraries) and are loaded by the server on demand. The dynamic loading feature is what distinguishes “C language” functions from “internal” functions --- the actual coding conventions are essentially the same for both. (Hence, the standard internal function library is a rich source of coding examples for user-defined C functions.)

Two different calling conventions are currently used for C functions. The newer “version 1” calling convention is indicated by writing a `PG_FUNCTION_INFO_V1()` macro call for the function, as illustrated below. Lack of such a macro indicates an old-style ("version 0") function. The language name specified in `CREATE FUNCTION` is `C` in either case. Old-style functions are now deprecated because of portability problems and lack of functionality, but they are still supported for compatibility reasons.

12.5.1. Dynamic Loading

The first time a user-defined function in a particular loadable object file is called in a backend session, the dynamic loader loads that object file into memory so that the function can be called. The `CREATE FUNCTION` for a user-defined C function must therefore specify two pieces of information for the function: the name of the loadable object file, and the C name (link symbol) of the specific function to call within that object file. If the C name is not explicitly specified then it is assumed to be the same as the SQL function name.

The following algorithm is used to locate the shared object file based on the name given in the `CREATE FUNCTION` command:

1. If the name is an absolute path, the given file is loaded.

2. If the name starts with the string `$libdir`, that part is replaced by the PostgreSQL package library directory name, which is determined at build time.

3. If the name does not contain a directory part, the file is searched for in the path specified by the configuration variable `dynamic_library_path`.

4. Otherwise (the file was not found in the path, or it contains a non-absolute directory part), the dynamic loader will try to take the name as given, which will most likely fail. (It is unreliable to depend on the current working directory.)

If this sequence does not work, the platform-specific shared library file name extension (often `.so`) is appended to the given name and this sequence is tried again. If that fails as well, the load will fail.

Note

The user ID the PostgreSQL server runs as must be able to traverse the path to the file you intend to load. Making the file or a higher-level directory not readable and/or not executable by the "postgres" user is a common mistake.

In any case, the file name that is given in the `CREATE FUNCTION` command is recorded literally in the system catalogs, so if the file needs to be loaded again the same procedure is applied.

Note

PostgreSQL will not compile a C function automatically. The object file must be compiled before it is referenced in a `CREATE FUNCTION` command. See Section 12.5.7, "Compiling and Linking Dynamically-Loaded Functions" for additional information.

Note

After it is used for the first time, a dynamically loaded object file is retained in memory. Future calls in the same session to the function(s) in that file will only incur the small overhead of a symbol table lookup. If you need to force a reload of an object file, for example after recompiling it, use the `LOAD` command or begin a fresh session.

It is recommended to locate shared libraries either relative to `$libdir` or through the dynamic library path. This simplifies version upgrades if the new installation is at a different location. The actual directory that `$libdir` stands for can be found out with the command `pg_config --pkglibdir`.

Note

Before PostgreSQL release 7.2, only exact absolute paths to object files could be specified in `CREATE FUNCTION`. This approach is now deprecated since it makes the function definition unnecessarily unportable. It's best to specify just the shared library name with no path nor extension, and let the search mechanism provide that information instead.

12.5.2. Base Types in C-Language Functions

Table 12.1, "Equivalent C Types" gives the C type required for parameters in the C functions that will be loaded into PostgreSQL. The "Defined In" column gives the header file that needs to be included to get the type definition. (The actual definition may be in a different file that is included by the listed file. It is recommended that users stick to the defined interface.) Note that you should always include `postgres.h` first in any source file, because it declares a number of things that you will need anyway.

Table 12.1. Equivalent C Types for Built-In PostgreSQL Types

SQL Type	C Type	Defined In
`abstime`	`AbsoluteTime`	`utils/nabstime.h`
`boolean`	`bool`	`postgres.h` (maybe compiler built-in)

SQL Type	C Type	Defined In
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	(compiler built-in)
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
smallint (int2)	int2 or int16	postgres.h
int2vector	int2vector*	postgres.h
integer (int4)	int4 or 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
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

Internally, PostgreSQL regards a base type as a “blob of memory”. The user-defined functions that you define over a type in turn define the way that PostgreSQL can operate on it. That is, PostgreSQL will only store and retrieve the data from disk and use your user-defined functions to input, process, and output the data. Base types can have one of three internal formats:

- pass by value, fixed-length
- pass by reference, fixed-length
- pass by reference, variable-length

By-value types can only be 1, 2 or 4 bytes in length (also 8 bytes, if `sizeof(Datum)` is 8 on your machine). You should be careful to define your types such that they will be the same size (in bytes) on

all architectures. For example, the `long` type is dangerous because it is 4 bytes on some machines and 8 bytes on others, whereas `int` type is 4 bytes on most Unix machines. A reasonable implementation of the `int4` type on Unix machines might be:

```
/* 4-byte integer, passed by value */
typedef int int4;
```

PostgreSQL automatically figures things out so that the integer types really have the size they advertise.

On the other hand, fixed-length types of any size may be passed by-reference. For example, here is a sample implementation of a PostgreSQL type:

```
/* 16-byte structure, passed by reference */
typedef struct
{
    double  x, y;
} Point;
```

Only pointers to such types can be used when passing them in and out of PostgreSQL functions. To return a value of such a type, allocate the right amount of memory with `palloc()`, fill in the allocated memory, and return a pointer to it. (Alternatively, you can return an input value of the same type by returning its pointer. *Never* modify the contents of a pass-by-reference input value, however.)

Finally, all variable-length types must also be passed by reference. All variable-length types must begin with a length field of exactly 4 bytes, and all data to be stored within that type must be located in the memory immediately following that length field. The length field is the total length of the structure (i.e., it includes the size of the length field itself). We can define the text type as follows:

```
typedef struct {
    int4 length;
    char data[1];
} text;
```

Obviously, the data field declared here is not long enough to hold all possible strings. Since it's impossible to declare a variable-size structure in C, we rely on the knowledge that the C compiler won't range-check array subscripts. We just allocate the necessary amount of space and then access the array as if it were declared the right length. (If this isn't a familiar trick to you, you may wish to spend some time with an introductory C programming textbook before delving deeper into PostgreSQL server programming.) When manipulating variable-length types, we must be careful to allocate the correct amount of memory and set the length field correctly. For example, if we wanted to store 40 bytes in a text structure, we might use a code fragment like this:

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
destination->length = VARHDRSZ + 40;
memcpy(destination->data, buffer, 40);
...
```

`VARHDRSZ` is the same as `sizeof(int4)`, but it's considered good style to use the macro `VARHDRSZ` to refer to the size of the overhead for a variable-length type.

Now that we've gone over all of the possible structures for base types, we can show some examples of real functions.

12.5.3. Version-0 Calling Conventions for C-Language Functions

We present the “old style” calling convention first --- although this approach is now deprecated, it's easier to get a handle on initially. In the version-0 method, the arguments and result of the C function are just declared in normal C style, but being careful to use the C representation of each SQL data type as shown above.

Here are some examples:

```
#include "postgres.h"
#include <string.h>

/* By Value */

int
add_one(int arg)
{
    return arg + 1;
}

/* By Reference, Fixed Length */

float8 *
add_one_float8(float8 *arg)
{
    float8    *result = (float8 *) palloc(sizeof(float8));

    *result = *arg + 1.0;

    return result;
}

Point *
makepoint(Point *pointx, Point *pointy)
{
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    return new_point;
}

/* By Reference, Variable Length */

text *
copytext(text *t)
{
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text *new_t = (text *) palloc(VARSIZE(t));
```

```
    VARATT_SIZEP(new_t) = VARSIZE(t);
    /*
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t)-VARHDRSZ);    /* how many bytes */
    return new_t;
}

text *
concat_text(text *arg1, text *arg2)
{
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    VARATT_SIZEP(new_text) = new_text_size;
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1)-VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1)-VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2)-VARHDRSZ);
    return new_text;
}
```

Supposing that the above code has been prepared in file `funcs.c` and compiled into a shared object, we could define the functions to PostgreSQL with commands like this:

```
CREATE FUNCTION add_one(int4) RETURNS int4
     AS 'PGROOT/tutorial/funcs' LANGUAGE C
     WITH (isStrict);

-- note overloading of SQL function name add_one()
CREATE FUNCTION add_one(float8) RETURNS float8
     AS 'PGROOT/tutorial/funcs',
        'add_one_float8'
     LANGUAGE C WITH (isStrict);

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'PGROOT/tutorial/funcs' LANGUAGE C
     WITH (isStrict);

CREATE FUNCTION copytext(text) RETURNS text
     AS 'PGROOT/tutorial/funcs' LANGUAGE C
     WITH (isStrict);

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS 'PGROOT/tutorial/funcs' LANGUAGE C
     WITH (isStrict);
```

Here *PGROOT* stands for the full path to the PostgreSQL source tree. (Better style would be to use just `'funcs'` in the `AS` clause, after having added *PGROOT*`/tutorial` to the search path. In any case, we may omit the system-specific extension for a shared library, commonly `.so` or `.sl`.)

Notice that we have specified the functions as "strict", meaning that the system should automatically assume a NULL result if any input value is NULL. By doing this, we avoid having to check for NULL inputs in the function code. Without this, we'd have to check for NULLs explicitly, for example by checking

for a null pointer for each pass-by-reference argument. (For pass-by-value arguments, we don't even have a way to check!)

Although this calling convention is simple to use, it is not very portable; on some architectures there are problems with passing smaller-than-int data types this way. Also, there is no simple way to return a NULL result, nor to cope with NULL arguments in any way other than making the function strict. The version-1 convention, presented next, overcomes these objections.

12.5.4. Version-1 Calling Conventions for C-Language Functions

The version-1 calling convention relies on macros to suppress most of the complexity of passing arguments and results. The C declaration of a version-1 function is always

```
Datum funcname(PG_FUNCTION_ARGS)
```

In addition, the macro call

```
PG_FUNCTION_INFO_V1(funcname);
```

must appear in the same source file (conventionally it's written just before the function itself). This macro call is not needed for `internal`-language functions, since PostgreSQL currently assumes all internal functions are version-1. However, it is *required* for dynamically-loaded functions.

In a version-1 function, each actual argument is fetched using a `PG_GETARG_`*`xxx`*`()` macro that corresponds to the argument's datatype, and the result is returned using a `PG_RETURN_`*`xxx`*`()` macro for the return type.

Here we show the same functions as above, coded in version-1 style:

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"

/* By Value */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* By Reference, Fixed Length */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
    /* The macros for FLOAT8 hide its pass-by-reference nature */
```

```
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* Here, the pass-by-reference nature of Point is not hidden */
    Point     *pointx = PG_GETARG_POINT_P(0);
    Point     *pointy = PG_GETARG_POINT_P(1);
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* By Reference, Variable Length */

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_P(0);
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text     *new_t = (text *) palloc(VARSIZE(t));
    VARATT_SIZEP(new_t) = VARSIZE(t);
    /*
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t)-VARHDRSZ);    /* how many bytes */
    PG_RETURN_TEXT_P(new_t);
}

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 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    VARATT_SIZEP(new_text) = new_text_size;
```

```
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1)-VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1)-VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2)-VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

The `CREATE FUNCTION` commands are the same as for the version-0 equivalents.

At first glance, the version-1 coding conventions may appear to be just pointless obscurantism. However, they do offer a number of improvements, because the macros can hide unnecessary detail. An example is that in coding `add_one_float8`, we no longer need to be aware that `float8` is a pass-by-reference type. Another example is that the `GETARG` macros for variable-length types hide the need to deal with fetching "toasted" (compressed or out-of-line) values. The old-style `copytext` and `concat_text` functions shown above are actually wrong in the presence of toasted values, because they don't call `pg_detoast_datum()` on their inputs. (The handler for old-style dynamically-loaded functions currently takes care of this detail, but it does so less efficiently than is possible for a version-1 function.)

One big improvement in version-1 functions is better handling of NULL inputs and results. The macro `PG_ARGISNULL(`*n*`)` allows a function to test whether each input is NULL (of course, doing this is only necessary in functions not declared "strict"). As with the `PG_GETARG_`*xxx*`()` macros, the input arguments are counted beginning at zero. Note that one should refrain from executing `PG_GETARG_`*xxx*`()` until one has verified that the argument isn't NULL. To return a NULL result, execute `PG_RETURN_NULL()`; this works in both strict and nonstrict functions.

The version-1 function call conventions make it possible to return "set" results and implement trigger functions and procedural-language call handlers. Version-1 code is also more portable than version-0, because it does not break ANSI C restrictions on function call protocol. For more details see `src/backend/utils/fmgr/README` in the source distribution.

12.5.5. Composite Types in C-Language Functions

Composite types do not have a fixed layout like C structures. Instances of a composite type may contain null fields. In addition, composite types that are part of an inheritance hierarchy may have different fields than other members of the same inheritance hierarchy. Therefore, PostgreSQL provides a procedural interface for accessing fields of composite types from C. As PostgreSQL processes a set of rows, each row will be passed into your function as an opaque structure of type `TUPLE`. Suppose we want to write a function to answer the query

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
FROM emp
WHERE name = 'Bill' OR name = 'Sam';
```

In the query above, we can define `c_overpaid` as:

```
#include "postgres.h"
#include "executor/executor.h"  /* for GetAttributeByName() */

bool
c_overpaid(TupleTableSlot *t, /* the current row of EMP */
           int32 limit)
{
    bool isnull;
    int32 salary;

    salary = DatumGetInt32(GetAttributeByName(t, "salary", &isnull));
```

```
    if (isnull)
        return (false);
    return salary > limit;
}

/* In version-1 coding, the above would look like this: */

PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
    TupleTableSlot  *t = (TupleTableSlot *) PG_GETARG_POINTER(0);
    int32            limit = PG_GETARG_INT32(1);
    bool isnull;
    int32 salary;

    salary = DatumGetInt32(GetAttributeByName(t, "salary", &isnull));
    if (isnull)
        PG_RETURN_BOOL(false);
    /* Alternatively, we might prefer to do PG_RETURN_NULL() for null
 salary */

    PG_RETURN_BOOL(salary > limit);
}
```

`GetAttributeByName` is the PostgreSQL system function that returns attributes out of the current row. It has three arguments: the argument of type `TupleTableSlot*` passed into the function, the name of the desired attribute, and a return parameter that tells whether the attribute is null. `GetAttributeByName` returns a `Datum` value that you can convert to the proper data type by using the appropriate `DatumGet`*`XXX`*`()` macro.

The following command lets PostgreSQL know about the `c_overpaid` function:

```
CREATE FUNCTION c_overpaid(emp, int4)
RETURNS bool
AS 'PGROOT/tutorial/funcs'
LANGUAGE C;
```

While there are ways to construct new rows or modify existing rows from within a C function, these are far too complex to discuss in this manual. Consult the backend source code for examples.

12.5.6. Writing Code

We now turn to the more difficult task of writing programming language functions. Be warned: this section of the manual will not make you a programmer. You must have a good understanding of C (including the use of pointers and the malloc memory manager) before trying to write C functions for use with PostgreSQL. While it may be possible to load functions written in languages other than C into PostgreSQL, this is often difficult (when it is possible at all) because other languages, such as FORTRAN and Pascal often do not follow the same *calling convention* as C. That is, other languages do not pass argument and return values between functions in the same way. For this reason, we will assume that your programming language functions are written in C.

The basic rules for building C functions are as follows:

- Use `pg_config --includedir-server` to find out where the PostgreSQL server header files are installed on your system (or the system that your users will be running on). This option is new with PostgreSQL 7.2. For PostgreSQL 7.1 you should use the option `--includedir`. (`pg_config` will exit with a non-zero status if it encounters an unknown option.) For releases prior to 7.1 you will have to guess, but since that was before the current calling conventions were introduced, it is unlikely that you want to support those releases.

- When allocating memory, use the PostgreSQL routines `palloc` and `pfree` instead of the corresponding C library routines `malloc` and `free`. The memory allocated by `palloc` will be freed automatically at the end of each transaction, preventing memory leaks.

- Always zero the bytes of your structures using `memset` or `bzero`. Several routines (such as the hash access method, hash join and the sort algorithm) compute functions of the raw bits contained in your structure. Even if you initialize all fields of your structure, there may be several bytes of alignment padding (holes in the structure) that may contain garbage values.

- Most of the internal PostgreSQL types are declared in `postgres.h`, while the function manager interfaces (`PG_FUNCTION_ARGS`, etc.) are in `fmgr.h`, so you will need to include at least these two files. For portability reasons it's best to include `postgres.h` *first*, before any other system or user header files. Including `postgres.h` will also include `elog.h` and `palloc.h` for you.

- Symbol names defined within object files must not conflict with each other or with symbols defined in the PostgreSQL server executable. You will have to rename your functions or variables if you get error messages to this effect.

- Compiling and linking your object code so that it can be dynamically loaded into PostgreSQL always requires special flags. See Section 12.5.7, “Compiling and Linking Dynamically-Loaded Functions” for a detailed explanation of how to do it for your particular operating system.

12.5.7. Compiling and Linking Dynamically-Loaded Functions

Before you are able to use your PostgreSQL extension functions written in C, they must be compiled and linked in a special way to produce a file that can be dynamically loaded by the server. To be precise, a *shared library* needs to be created.

For more information you should read the documentation of your operating system, in particular the manual pages for the C compiler, `cc`, and the link editor, `ld`. In addition, the PostgreSQL source code contains several working examples in the `contrib` directory. If you rely on these examples you will make your modules dependent on the availability of the PostgreSQL source code, however.

Creating shared libraries is generally analogous to linking executables: first the source files are compiled into object files, then the object files are linked together. The object files need to be created as *position-independent code* (PIC), which conceptually means that they can be placed at an arbitrary location in memory when they are loaded by the executable. (Object files intended for executables are usually not compiled that way.) The command to link a shared library contains special flags to distinguish it from linking an executable. --- At least this is the theory. On some systems the practice is much uglier.

In the following examples we assume that your source code is in a file `foo.c` and we will create a shared library `foo.so`. The intermediate object file will be called `foo.o` unless otherwise noted. A shared library can contain more than one object file, but we only use one here.

BSD/OS

The compiler flag to create PIC is `-fpic`. The linker flag to create shared libraries is `-shared`.

```
gcc -fpic -c foo.c
ld -shared -o foo.so foo.o
```

This is applicable as of version 4.0 of BSD/OS.

FreeBSD

The compiler flag to create PIC is `-fpic`. To create shared libraries the compiler flag is `-shared`.

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

This is applicable as of version 3.0 of FreeBSD.

HP-UX

The compiler flag of the system compiler to create PIC is `+z`. When using GCC it's `-fpic`. The linker flag for shared libraries is `-b`. So

```
cc +z -c foo.c
```

or

```
gcc -fpic -c foo.c
```

and then

```
ld -b -o foo.sl foo.o
```

HP-UX uses the extension `.sl` for shared libraries, unlike most other systems.

IRIX

PIC is the default, no special compiler options are necessary. The linker option to produce shared libraries is `-shared`.

```
cc -c foo.c
ld -shared -o foo.so foo.o
```

Linux

The compiler flag to create PIC is `-fpic`. On some platforms in some situations `-fPIC` must be used if `-fpic` does not work. Refer to the GCC manual for more information. The compiler flag to create a shared library is `-shared`. A complete example looks like this:

```
cc -fpic -c foo.c
cc -shared -o foo.so foo.o
```

NetBSD

The compiler flag to create PIC is `-fpic`. For ELF systems, the compiler with the flag `-shared` is used to link shared libraries. On the older non-ELF systems, `ld -Bshareable` is used.

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

The compiler flag to create PIC is `-fpic`. `ld -Bshareable` is used to link shared libraries.

```
gcc -fpic -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

The compiler flag to create PIC is `-KPIC` with the Sun compiler and `-fpic` with GCC. To link shared libraries, the compiler option is `-G` with either compiler or alternatively `-shared` with GCC.

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

or

```
gcc -fpic -c foo.c
gcc -G -o foo.so foo.o
```

Tru64 UNIX

PIC is the default, so the compilation command is the usual one. `ld` with special options is used to do the linking:

```
cc -c foo.c
ld -shared -expect_unresolved '*' -o foo.so foo.o
```

The same procedure is used with GCC instead of the system compiler; no special options are required.

UnixWare

The compiler flag to create PIC is `-K PIC` with the SCO compiler and `-fpic` with GCC. To link shared libraries, the compiler option is `-G` with the SCO compiler and `-shared` with 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
```

Tip

If you want to package your extension modules for wide distribution you should consider using GNU Libtool[1] for building shared libraries. It encapsulates the platform differences into a general and powerful interface. Serious packaging also requires considerations about library versioning, symbol resolution methods, and other issues.

The resulting shared library file can then be loaded into PostgreSQL. When specifying the file name to the `CREATE FUNCTION` command, one must give it the name of the shared library file, not the intermediate object file. Note that the system's standard shared-library extension (usually `.so` or `.sl`) can be omitted from the `CREATE FUNCTION` command, and normally should be omitted for best portability.

Refer back to Section 12.5.1, “Dynamic Loading” about where the server expects to find the shared library files.

12.6. Function Overloading

[1] http://www.gnu.org/software/libtool/

More than one function may be defined with the same SQL name, so long as the arguments they take are different. In other words, function names can be *overloaded*. When a query is executed, the server will determine which function to call from the data types and the number of the provided arguments. Overloading can also be used to simulate functions with a variable number of arguments, up to a finite maximum number.

A function may also have the same name as an attribute. In the case that there is an ambiguity between a function on a complex type and an attribute of the complex type, the attribute will always be used.

When creating a family of overloaded functions, one should be careful not to create ambiguities. For instance, given the functions

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

it is not immediately clear which function would be called with some trivial input like `test(1, 1.5)`. The currently implemented resolution rules are described in the *User's Guide*, but it is unwise to design a system that subtly relies on this behavior.

When overloading C language functions, there is an additional constraint: The C name of each function in the family of overloaded functions must be different from the C names of all other functions, either internal or dynamically loaded. If this rule is violated, the behavior is not portable. You might get a run-time linker error, or one of the functions will get called (usually the internal one). The alternative form of the `AS` clause for the SQL `CREATE FUNCTION` command decouples the SQL function name from the function name in the C source code. E.g.,

```
CREATE FUNCTION test(int) RETURNS int
    AS 'filename', 'test_1arg'
    LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
    AS 'filename', 'test_2arg'
    LANGUAGE C;
```

The names of the C functions here reflect one of many possible conventions.

Prior to PostgreSQL 7.0, this alternative syntax did not exist. There is a trick to get around the problem, by defining a set of C functions with different names and then define a set of identically-named SQL function wrappers that take the appropriate argument types and call the matching C function.

12.7. Procedural Language Handlers

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 section describes how a language call handler can be written. This is not a common task, in fact, it has only been done a handful of times in the history of PostgreSQL, but the topic naturally belongs in this chapter, and the material might give some insight into the extensible nature of the PostgreSQL system.

The call handler for a procedural language is a "normal" function, which must be written in a compiled language such as C and registered with PostgreSQL as taking no arguments and returning the `opaque` type, a placeholder for unspecified or undefined types. This prevents the call handler from being called directly as a function from queries. (However, arguments may be supplied in the actual call to the handler when a function in the language offered by the handler is to be executed.)

Note

In PostgreSQL 7.1 and later, call handlers must adhere to the “version 1” function manager interface, not the old-style interface.

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` struct, 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` struct 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 `pg_proc` entry and to analyze the argument and return types of the called procedure. The AS clause from the `CREATE FUNCTION` of the procedure will be found in the `prosrc` attribute of the `pg_proc` table entry. This may be the source text in the procedural language itself (like for PL/Tcl), 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 PL 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 be careful 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 context so that it can cache function definition information across queries.

When a PL function is invoked as a trigger, no explicit arguments are passed, but the `FunctionCallInfoData`'s `context` field points at a `TriggerData` node, rather than being NULL as it is in a plain function call. A language handler should provide mechanisms for PL functions to get at the trigger information.

This is a template for a PL handler written in C:

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
#include "utils/elog.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. See Section 12.5, “C Language Functions” for information on how to compile it into a loadable module.

The following commands then register the sample procedural language:

```
CREATE FUNCTION plsample_call_handler () RETURNS opaque
    AS '/usr/local/pgsql/lib/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

Chapter 13. Extending SQL: Types

As previously mentioned, there are two kinds of types in PostgreSQL: base types (defined in a programming language) and composite types. This chapter describes how to define new base types.

The examples in this section can be found in `complex.sql` and `complex.c` in the tutorial directory. Composite examples are in `funcs.sql`.

A user-defined type must always have input and output functions. These functions determine how the type appears in strings (for input by the user and output to the user) and how the type is organized in memory. The input function takes a null-terminated character string as its input and returns the internal (in memory) representation of the type. The output function takes the internal representation of the type and returns a null-terminated character string.

Suppose we want to define a complex type which represents complex numbers. Naturally, we would choose to represent a complex in memory as the following C structure:

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

and a string of the form `(x,y)` as the external string representation.

The functions are usually not hard to write, especially the output function. However, there are a number of points to remember:

- When defining your external (string) representation, remember that you must eventually write a complete and robust parser for that representation as your input function!

  For instance:

  ```
  Complex *
  complex_in(char *str)
  {
      double x, y;
      Complex *result;
      if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2) {
          elog(ERROR, "complex_in: error in parsing %s", str);
          return NULL;
      }
      result = (Complex *)palloc(sizeof(Complex));
      result->x = x;
      result->y = y;
      return (result);
  }
  ```

  The output function can simply be:

  ```
  char *
  complex_out(Complex *complex)
  {
      char *result;
      if (complex == NULL)
          return(NULL);
  ```

```
    result = (char *) palloc(60);
    sprintf(result, "(%g,%g)", complex->x, complex->y);
    return(result);
}
```

- You should try to make the input and output functions inverses of each other. If you do not, you will have severe problems when you need to dump your data into a file and then read it back in (say, into someone else's database on another computer). This is a particularly common problem when floating-point numbers are involved.

To define the `complex` type, we need to create the two user-defined functions `complex_in` and `complex_out` before creating the type:

```
CREATE FUNCTION complex_in(opaque)
    RETURNS complex
    AS 'PGROOT/tutorial/complex'
    LANGUAGE C;

CREATE FUNCTION complex_out(opaque)
    RETURNS opaque
    AS 'PGROOT/tutorial/complex'
    LANGUAGE C;
```

Finally, we can declare the data type:

```
CREATE TYPE complex (
    internallength = 16,
    input = complex_in,
    output = complex_out
);
```

As discussed earlier, PostgreSQL fully supports arrays of base types. Additionally, PostgreSQL supports arrays of user-defined types as well. When you define a type, PostgreSQL automatically provides support for arrays of that type. For historical reasons, the array type has the same name as the user-defined type with the underscore character _ prepended.

Composite types do not need any function defined on them, since the system already understands what they look like inside.

If the values of your datatype might exceed a few hundred bytes in size (in internal form), you should be careful to mark them TOAST-able. To do this, the internal representation must follow the standard layout for variable-length data: the first four bytes must be an `int32` containing the total length in bytes of the datum (including itself). Then, all your functions that accept values of the type must be careful to call `pg_detoast_datum()` on the supplied values --- after checking that the value is not NULL, if your function is not strict. Finally, select the appropriate storage option when giving the `CREATE TYPE` command.

Chapter 14. Extending SQL: Operators

14.1. Introduction

PostgreSQL supports left unary, right unary, and binary operators. Operators can be overloaded; that is, the same operator name can be used for different operators that have different numbers and types of operands. If there is an ambiguous situation and the system cannot determine the correct operator to use, it will return an error. You may have to type-cast the left and/or right operands to help it understand which operator you meant to use.

Every operator is "syntactic sugar" for a call to an underlying function that does the real work; so you must first create the underlying function before you can create the operator. However, an operator is *not merely* syntactic sugar, because it carries additional information that helps the query planner optimize queries that use the operator. Much of this chapter will be devoted to explaining that additional information.

14.2. Example

Here is an example of creating an operator for adding two complex numbers. We assume we've already created the definition of type `complex` (see Chapter 13, *Extending SQL: Types*). First we need a function that does the work, then we can define the operator:

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'PGROOT/tutorial/complex'
    LANGUAGE C;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

Now we can do:

```
SELECT (a + b) AS c FROM test_complex;

        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
```

We've shown how to create a binary operator here. To create unary operators, just omit one of `leftarg` (for left unary) or `rightarg` (for right unary). The `procedure` clause and the argument clauses are the only required items in `CREATE OPERATOR`. The `commutator` clause shown in the example is an optional hint to the query optimizer. Further details about `commutator` and other optimizer hints appear below.

14.3. Operator Optimization Information

Author

Written by Tom Lane.

A PostgreSQL operator definition can include several optional clauses that tell the system useful things about how the operator behaves. These clauses should be provided whenever appropriate, because they can make for considerable speedups in execution of queries that use the operator. But if you provide them, you must be sure that they are right! Incorrect use of an optimization clause can result in backend crashes, subtly wrong output, or other Bad Things. You can always leave out an optimization clause if you are not sure about it; the only consequence is that queries might run slower than they need to.

Additional optimization clauses might be added in future versions of PostgreSQL. The ones described here are all the ones that release 7.2.8 understands.

14.3.1. COMMUTATOR

The `COMMUTATOR` clause, if provided, names an operator that is the commutator of the operator being defined. We say that operator A is the commutator of operator B if (x A y) equals (y B x) for all possible input values x, y. Notice that B is also the commutator of A. For example, operators < and > for a particular data type are usually each others' commutators, and operator + is usually commutative with itself. But operator − is usually not commutative with anything.

The left operand type of a commuted operator is the same as the right operand type of its commutator, and vice versa. So the name of the commutator operator is all that PostgreSQL needs to be given to look up the commutator, and that's all that needs to be provided in the `COMMUTATOR` clause.

When you are defining a self-commutative operator, you just do it. When you are defining a pair of commutative operators, things are a little trickier: how can the first one to be defined refer to the other one, which you haven't defined yet? There are two solutions to this problem:

- One way is to omit the `COMMUTATOR` clause in the first operator that you define, and then provide one in the second operator's definition. Since PostgreSQL knows that commutative operators come in pairs, when it sees the second definition it will automatically go back and fill in the missing `COMMUTATOR` clause in the first definition.
- The other, more straightforward way is just to include `COMMUTATOR` clauses in both definitions. When PostgreSQL processes the first definition and realizes that `COMMUTATOR` refers to a non-existent operator, the system will make a dummy entry for that operator in the system catalog. This dummy entry will have valid data only for the operator name, left and right operand types, and result type, since that's all that PostgreSQL can deduce at this point. The first operator's catalog entry will link to this dummy entry. Later, when you define the second operator, the system updates the dummy entry with the additional information from the second definition. If you try to use the dummy operator before it's been filled in, you'll just get an error message. (Note: This procedure did not work reliably in PostgreSQL versions before 6.5, but it is now the recommended way to do things.)

14.3.2. NEGATOR

The `NEGATOR` clause, if provided, names an operator that is the negator of the operator being defined. We say that operator A is the negator of operator B if both return Boolean results and (x A y) equals NOT (x B y) for all possible inputs x, y. Notice that B is also the negator of A. For example, < and >= are a negator pair for most data types. An operator can never validly be its own negator.

Unlike commutators, a pair of unary operators could validly be marked as each others' negators; that would mean (A x) equals NOT (B x) for all x, or the equivalent for right unary operators.

An operator's negator must have the same left and/or right operand types as the operator itself, so just as with `COMMUTATOR`, only the operator name need be given in the `NEGATOR` clause.

Providing a negator is very helpful to the query optimizer since it allows expressions like NOT (x = y) to be simplified into x <> y. This comes up more often than you might think, because NOTs can be inserted as a consequence of other rearrangements.

Pairs of negator operators can be defined using the same methods explained above for commutator pairs.

14.3.3. RESTRICT

The `RESTRICT` clause, if provided, names a restriction selectivity estimation function for the operator (note that this is a function name, not an operator name). `RESTRICT` clauses only make sense for binary operators that return `boolean`. The idea behind a restriction selectivity estimator is to guess what fraction of the rows in a table will satisfy a `WHERE`-clause condition of the form

```
column OP constant
```

for the current operator and a particular constant value. This assists the optimizer by giving it some idea of how many rows will be eliminated by `WHERE` clauses that have this form. (What happens if the constant is on the left, you may be wondering? Well, that's one of the things that `COMMUTATOR` is for...)

Writing new restriction selectivity estimation functions is far beyond the scope of this chapter, but fortunately you can usually just use one of the system's standard estimators for many of your own operators. These are the standard restriction estimators:

`eqsel` for =
`neqsel` for <>
`scalarltsel` for < or <=
`scalargtsel` for > or >=
It might seem a little odd that these are the categories, but they make sense if you think about it. = will typically accept only a small fraction of the rows in a table; <> will typically reject only a small fraction. < will accept a fraction that depends on where the given constant falls in the range of values for that table column (which, it just so happens, is information collected by `ANALYZE` and made available to the selectivity estimator). <= will accept a slightly larger fraction than < for the same comparison constant, but they're close enough to not be worth distinguishing, especially since we're not likely to do better than a rough guess anyhow. Similar remarks apply to > and >=.

You can frequently get away with using either `eqsel` or `neqsel` for operators that have very high or very low selectivity, even if they aren't really equality or inequality. For example, the approximate-equality geometric operators use `eqsel` on the assumption that they'll usually only match a small fraction of the entries in a table.

You can use `scalarltsel` and `scalargtsel` for comparisons on data types that have some sensible means of being converted into numeric scalars for range comparisons. If possible, add the data type to those understood by the routine `convert_to_scalar()` in `src/backend/utils/adt/selfuncs.c`. (Eventually, this routine should be replaced by per-data-type functions identified through a column of the `pg_type` system catalog; but that hasn't happened yet.) If you do not do this, things will still work, but the optimizer's estimates won't be as good as they could be.

There are additional selectivity functions designed for geometric operators in `src/backend/utils/adt/geo_selfuncs.c`: `areasel`, `positionsel`, and `contsel`. At this writing these are just stubs, but you may want to use them (or even better, improve them) anyway.

14.3.4. JOIN

The JOIN clause, if provided, names a join selectivity estimation function for the operator (note that this is a function name, not an operator name). JOIN clauses only make sense for binary operators that return boolean. The idea behind a join selectivity estimator is to guess what fraction of the rows in a pair of tables will satisfy a WHERE-clause condition of the form

```
table1.column1 OP table2.column2
```

for the current operator. As with the RESTRICT clause, this helps the optimizer very substantially by letting it figure out which of several possible join sequences is likely to take the least work.

As before, this chapter will make no attempt to explain how to write a join selectivity estimator function, but will just suggest that you use one of the standard estimators if one is applicable:

eqjoinsel for =
neqjoinsel for <>
scalarltjoinsel for < or <=
scalargtjoinsel for > or >=
areajoinsel for 2D area-based comparisons
positionjoinsel for 2D position-based comparisons
contjoinsel for 2D containment-based comparisons

14.3.5. HASHES

The HASHES clause, if present, tells the system that it is OK to use the hash join method for a join based on this operator. HASHES only makes sense for binary operators that return boolean, and in practice the operator had better be equality for some data type.

The assumption underlying hash join is that the join operator can only return true for pairs of left and right values that hash to the same hash code. If two values get put in different hash buckets, the join will never compare them at all, implicitly assuming that the result of the join operator must be false. So it never makes sense to specify HASHES for operators that do not represent equality.

In fact, logical equality is not good enough either; the operator had better represent pure bitwise equality, because the hash function will be computed on the memory representation of the values regardless of what the bits mean. For example, equality of time intervals is not bitwise equality; the interval equality operator considers two time intervals equal if they have the same duration, whether or not their endpoints are identical. What this means is that a join using = between interval fields would yield different results if implemented as a hash join than if implemented another way, because a large fraction of the pairs that should match will hash to different values and will never be compared by the hash join. But if the optimizer chose to use a different kind of join, all the pairs that the equality operator says are equal will be found. We don't want that kind of inconsistency, so we don't mark interval equality as hashable.

There are also machine-dependent ways in which a hash join might fail to do the right thing. For example, if your data type is a structure in which there may be uninteresting pad bits, it's unsafe to mark the equality operator HASHES. (Unless, perhaps, you write your other operators to ensure that the unused bits are always zero.) Another example is that the floating-point data types are unsafe for hash joins. On machines that meet the IEEE floating-point standard, minus zero and plus zero are different values (different bit patterns) but they are defined to compare equal. So, if the equality operator on floating-point data types were marked HASHES, a minus zero and a plus zero would probably not be matched up by a hash join, but they would be matched up by any other join process.

The bottom line is that you should probably only use HASHES for equality operators that are (or could be) implemented by memcmp().

14.3.6. SORT1 and SORT2

The SORT clauses, if present, tell the system that it is permissible to use the merge join method for a join based on the current operator. Both must be specified if either is. The current operator must be equality for some pair of data types, and the SORT1 and SORT2 clauses name the ordering operator ("<" operator) for the left and right-side data types respectively.

Merge join is based on the idea of sorting the left and righthand tables into order and then scanning them in parallel. So, both data types must be capable of being fully ordered, and the join operator must be one that can only succeed for pairs of values that fall at the "same place" in the sort order. In practice this means that the join operator must behave like equality. But unlike hash join, where the left and right data types had better be the same (or at least bitwise equivalent), it is possible to merge-join two distinct data types so long as they are logically compatible. For example, the int2-versus-int4 equality operator is merge-joinable. We only need sorting operators that will bring both data types into a logically compatible sequence.

When specifying merge-sort operators, the current operator and both referenced operators must return boolean; the SORT1 operator must have both input data types equal to the current operator's left operand type, and the SORT2 operator must have both input data types equal to the current operator's right operand type. (As with COMMUTATOR and NEGATOR, this means that the operator name is sufficient to specify the operator, and the system is able to make dummy operator entries if you happen to define the equality operator before the other ones.)

In practice you should only write SORT clauses for an = operator, and the two referenced operators should always be named <. Trying to use merge join with operators named anything else will result in hopeless confusion, for reasons we'll see in a moment.

There are additional restrictions on operators that you mark merge-joinable. These restrictions are not currently checked by CREATE OPERATOR, but a merge join may fail at run time if any are not true:

- The merge-joinable equality operator must have a commutator (itself if the two data types are the same, or a related equality operator if they are different).
- There must be < and > ordering operators having the same left and right operand data types as the merge-joinable operator itself. These operators *must* be named < and >; you do not have any choice in the matter, since there is no provision for specifying them explicitly. Note that if the left and right data types are different, neither of these operators is the same as either SORT operator. But they had better order the data values compatibly with the SORT operators, or the merge join will fail to work.

Chapter 15. Extending SQL: Aggregates

Aggregate functions in PostgreSQL are expressed as *state values* and *state transition functions*. That is, an aggregate can be defined in terms of state that is modified whenever an input item is processed. To define a new aggregate function, one selects a data type for the state value, an initial value for the state, and a state transition function. The state transition function is just an ordinary function that could also be used outside the context of the aggregate. A *final function* can also be specified, in case the desired output of the aggregate is different from the data that needs to be kept in the running state value.

Thus, in addition to the input and result data types seen by a user of the aggregate, there is an internal state-value data type that may be different from both the input and result types.

If we define an aggregate that does not use a final function, we have an aggregate that computes a running function of the column values from each row. `Sum` is an example of this kind of aggregate. `Sum` starts at zero and always adds the current row's value to its running total. For example, if we want to make a `sum` aggregate to work on a data type for complex numbers, we only need the addition function for that data type. The aggregate definition is:

```
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)
```

(In practice, we'd just name the aggregate `sum`, and rely on PostgreSQL to figure out which kind of sum to apply to a column of type `complex`.)

The above definition of `sum` will return zero (the initial state condition) if there are no non-null input values. Perhaps we want to return NULL in that case instead --- the SQL standard expects `sum` to behave that way. We can do this simply by omitting the `initcond` phrase, so that the initial state condition is NULL. Ordinarily this would mean that the `sfunc` would need to check for a NULL state-condition input, but for `sum` and some other simple aggregates like `max` and `min`, it's sufficient to insert the first non-null input value into the state variable and then start applying the transition function at the second non-null input value. PostgreSQL will do that automatically if the initial condition is NULL and the transition function is marked “strict” (i.e., not to be called for NULL inputs).

Another bit of default behavior for a “strict” transition function is that the previous state value is retained unchanged whenever a NULL input value is encountered. Thus, NULLs are ignored. If you need some other behavior for NULL inputs, just define your transition function as non-strict, and code it to test for NULL inputs and do whatever is needed.

`Avg` (average) is a more complex example of an aggregate. It requires two pieces of running state: the sum of the inputs and the count of the number of inputs. The final result is obtained by dividing these quantities. Average is typically implemented by using a two-element array as the transition state value. For example, the built-in implementation of `avg(float8)` looks like:

```
CREATE AGGREGATE avg (
    sfunc = float8_accum,
    basetype = float8,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0}'
);
```

For further details see the description of the `CREATE AGGREGATE` command in the *Reference Manual*.

Chapter 16. The Rule System

Author

Written by Jan Wieck. Updates for 7.1 by Tom Lane.

16.1. Introduction

Production rule systems are conceptually simple, but there are many subtle points involved in actually using them. Some of these points and the theoretical foundations of the PostgreSQL rule system can be found in [ston90b].

Some other database systems define active database rules. These are usually stored procedures and triggers and are implemented in PostgreSQL as functions and triggers.

The query rewrite rule system (the *rule system* from now on) is totally different from stored procedures and triggers. It modifies queries to take rules into consideration, and then passes the modified query to the query planner for planning and execution. It is very powerful, and can be used for many things such as query language procedures, views, and versions. The power of this rule system is discussed in [ong90] as well as [ston90b].

16.2. What is a Query Tree?

To understand how the rule system works it is necessary to know when it is invoked and what its input and results are.

The rule system is located between the query parser and the planner. It takes the output of the parser, one query tree, and the rewrite rules from the `pg_rewrite` catalog, which are query trees too with some extra information, and creates zero or many query trees as result. So its input and output are always things the parser itself could have produced and thus, anything it sees is basically representable as an SQL statement.

Now what is a query tree? It is an internal representation of an SQL statement where the single parts that built it are stored separately. These query trees are visible when starting the PostgreSQL backend with debug level 4 and typing queries into the interactive backend interface. The rule actions in the `pg_rewrite` system catalog are also stored as query trees. They are not formatted like the debug output, but they contain exactly the same information.

Reading a query tree requires some experience and it was a hard time when I started to work on the rule system. I can remember that I was standing at the coffee machine and I saw the cup in a target list, water and coffee powder in a range table and all the buttons in a qualification expression. Since SQL representations of query trees are sufficient to understand the rule system, this document will not teach how to read them. It might help to learn it and the naming conventions are required in the later following descriptions.

16.2.1. The Parts of a Query tree

When reading the SQL representations of the query trees in this document it is necessary to be able to identify the parts the statement is broken into when it is in the query tree structure. The parts of a query tree are

the command type

This is a simple value telling which command (SELECT, INSERT, UPDATE, DELETE) produced the parse tree.

the range table

The range table is a list of relations that are used in the query. In a SELECT statement these are the relations given after the FROM keyword.

Every range table entry identifies a table or view and tells by which name it is called in the other parts of the query. In the query tree the range table entries are referenced by index rather than by name, so here it doesn't matter if there are duplicate names as it would in an SQL statement. This can happen after the range tables of rules have been merged in. The examples in this document will not have this situation.

the result relation

This is an index into the range table that identifies the relation where the results of the query go.

SELECT queries normally don't have a result relation. The special case of a SELECT INTO is mostly identical to a CREATE TABLE, INSERT ... SELECT sequence and is not discussed separately here.

On INSERT, UPDATE and DELETE queries the result relation is the table (or view!) where the changes take effect.

the target list

The target list is a list of expressions that define the result of the query. In the case of a SELECT, the expressions are what builds the final output of the query. They are the expressions between the SELECT and the FROM keywords. (* is just an abbreviation for all the attribute names of a relation. It is expanded by the parser into the individual attributes, so the rule system never sees it.)

DELETE queries don't need a target list because they don't produce any result. In fact the planner will add a special CTID entry to the empty target list. But this is after the rule system and will be discussed later. For the rule system the target list is empty.

In INSERT queries the target list describes the new rows that should go into the result relation. It is the expressions in the VALUES clause or the ones from the SELECT clause in INSERT ... SELECT. Missing columns of the result relation will be filled in by the planner with a constant NULL expression.

In UPDATE queries, the target list describes the new rows that should replace the old ones. In the rule system, it contains just the expressions from the SET attribute = expression part of the query. The planner will add missing columns by inserting expressions that copy the values from the old row into the new one. And it will add the special CTID entry just as for DELETE too.

Every entry in the target list contains an expression that can be a constant value, a variable pointing to an attribute of one of the relations in the range table, a parameter, or an expression tree made of function calls, constants, variables, operators etc.

the qualification

The query's qualification is an expression much like one of those contained in the target list entries. The result value of this expression is a Boolean that tells if the operation (INSERT, UPDATE, DELETE or SELECT) for the final result row should be executed or not. It is the WHERE clause of an SQL statement.

the join tree

The query's join tree shows the structure of the FROM clause. For a simple query like SELECT FROM a, b, c the join tree is just a list of the FROM items, because we are allowed to join them in any order. But when JOIN expressions --- particularly outer joins --- are used, we have to join in the order shown

by the joins. The join tree shows the structure of the JOIN expressions. The restrictions associated with particular JOIN clauses (from ON or USING expressions) are stored as qualification expressions attached to those join tree nodes. It turns out to be convenient to store the top-level WHERE expression as a qualification attached to the top-level join tree item, too. So really the join tree represents both the FROM and WHERE clauses of a SELECT.

the others

The other parts of the query tree like the ORDER BY clause aren't of interest here. The rule system substitutes entries there while applying rules, but that doesn't have much to do with the fundamentals of the rule system.

16.3. Views and the Rule System

16.3.1. Implementation of Views in PostgreSQL

Views in PostgreSQL are implemented using the rule system. In fact there is absolutely no difference between a

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

compared against the two commands

```
CREATE TABLE myview (same attribute list as for mytab);
CREATE RULE "_RETmyview" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

because this is exactly what the CREATE VIEW command does internally. This has some side effects. One of them is that the information about a view in the PostgreSQL system catalogs is exactly the same as it is for a table. So for the query parser, there is absolutely no difference between a table and a view. They are the same thing - relations. That is the important one for now.

16.3.2. How SELECT Rules Work

Rules ON SELECT are applied to all queries as the last step, even if the command given is an INSERT, UPDATE or DELETE. And they have different semantics from the others in that they modify the parse tree in place instead of creating a new one. So SELECT rules are described first.

Currently, there can be only one action in an ON SELECT rule, and it must be an unconditional SELECT action that is INSTEAD. This restriction was required to make rules safe enough to open them for ordinary users and it restricts rules ON SELECT to real view rules.

The examples for this document are two join views that do some calculations and some more views using them in turn. One of the two first views is customized later by adding rules for INSERT, UPDATE and DELETE operations so that the final result will be a view that behaves like a real table with some magic functionality. It is not such a simple example to start from and this makes things harder to get into. But it's better to have one example that covers all the points discussed step by step rather than having many different ones that might mix up in mind.

The database needed to play with the examples is named `al_bundy`. You'll see soon why this is the database name. And it needs the procedural language PL/pgSQL installed, because we need a little min() function returning the lower of 2 integer values. We create that as

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS '
    BEGIN
```

```
        IF $1 < $2 THEN
            RETURN $1;
        END IF;
        RETURN $2;
    END;
' LANGUAGE plpgsql;
```

The real tables we need in the first two rule system descriptions are these:

```
CREATE TABLE shoe_data (
    shoename   char(10),      -- primary key
    sh_avail   integer,       -- available # of pairs
    slcolor    char(10),      -- preferred shoelace color
    slminlen   float,         -- miminum shoelace length
    slmaxlen   float,         -- maximum shoelace length
    slunit     char(8)        -- length unit
);

CREATE TABLE shoelace_data (
    sl_name    char(10),      -- primary key
    sl_avail   integer,       -- available # of pairs
    sl_color   char(10),      -- shoelace color
    sl_len     float,         -- shoelace length
    sl_unit    char(8)        -- length unit
);

CREATE TABLE unit (
    un_name    char(8),       -- the primary key
    un_fact    float          -- factor to transform to cm
);
```

I think most of us wear shoes and can realize that this is really useful data. Well there are shoes out in the world that don't require shoelaces, but this doesn't make Al's life easier and so we ignore it.

The views are created as

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
```

```
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           min(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
     WHERE rsl.sl_color = rsh.slcolor
       AND rsl.sl_len_cm >= rsh.slminlen_cm
       AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

The CREATE VIEW command for the `shoelace` view (which is the simplest one we have) will create a relation shoelace and an entry in `pg_rewrite` that tells that there is a rewrite rule that must be applied whenever the relation shoelace is referenced in a query's range table. The rule has no rule qualification (discussed later, with the non SELECT rules, since SELECT rules currently cannot have them) and it is INSTEAD. Note that rule qualifications are not the same as query qualifications! The rule's action has a query qualification.

The rule's action is one query tree that is a copy of the SELECT statement in the view creation command.

Note

The two extra range table entries for NEW and OLD (named *NEW* and *CURRENT* for historical reasons in the printed query tree) you can see in the `pg_rewrite` entry aren't of interest for SELECT rules.

Now we populate `unit`, `shoe_data` and `shoelace_data` and Al types the first SELECT in his life:

```
al_bundy=> INSERT INTO unit VALUES ('cm', 1.0);
al_bundy=> INSERT INTO unit VALUES ('m', 100.0);
al_bundy=> INSERT INTO unit VALUES ('inch', 2.54);
al_bundy=>
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh1', 2, 'black', 70.0, 90.0, 'cm');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh2', 0, 'black', 30.0, 40.0, 'inch');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh4', 3, 'brown', 40.0, 50.0, 'inch');
al_bundy=>
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl1', 5, 'black', 80.0, 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl2', 6, 'black', 100.0, 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl3', 0, 'black', 35.0 , 'inch');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl4', 8, 'black', 40.0 , 'inch');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl5', 4, 'brown', 1.0 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
```

```
al_bundy->     ('sl6', 0, 'brown', 0.9 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl7', 7, 'brown', 60 , 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl8', 1, 'brown', 40 , 'inch');
al_bundy=>
al_bundy=> SELECT * FROM shoelace;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       7|brown     |    60|cm      |       60
sl3       |       0|black     |    35|inch    |     88.9
sl4       |       8|black     |    40|inch    |    101.6
sl8       |       1|brown     |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |       0|brown     |   0.9|m       |       90
(8 rows)
```

It's the simplest SELECT Al can do on our views, so we take this to explain the basics of view rules. The `SELECT * FROM shoelace` was interpreted by the parser and produced the parse tree

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

and this is given to the rule system. The rule system walks through the range table and checks if there are rules in `pg_rewrite` for any relation. When processing the range table entry for `shoelace` (the only one up to now) it finds the rule `_RETshoelace` with the parse tree

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       float8mul(s.sl_len, u.un_fact) AS sl_len_cm
  FROM shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u
 WHERE bpchareq(s.sl_unit, u.un_name);
```

Note that the parser changed the calculation and qualification into calls to the appropriate functions. But in fact this changes nothing.

To expand the view, the rewriter simply creates a subselect range-table entry containing the rule's action parse tree, and substitutes this range table entry for the original one that referenced the view. The resulting rewritten parse tree is almost the same as if Al had typed

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM (SELECT s.sl_name,
               s.sl_avail,
               s.sl_color,
               s.sl_len,
               s.sl_unit,
               s.sl_len * u.un_fact AS sl_len_cm
          FROM shoelace_data s, unit u
```

```
          WHERE s.sl_unit = u.un_name) shoelace;
```

There is one difference however: the sub-query's range table has two extra entries shoelace *OLD*, shoelace *NEW*. These entries don't participate directly in the query, since they aren't referenced by the sub-query's join tree or target list. The rewriter uses them to store the access permission check info that was originally present in the range-table entry that referenced the view. In this way, the executor will still check that the user has proper permissions to access the view, even though there's no direct use of the view in the rewritten query.

That was the first rule applied. The rule system will continue checking the remaining range-table entries in the top query (in this example there are no more), and it will recursively check the range-table entries in the added sub-query to see if any of them reference views. (But it won't expand *OLD* or *NEW* --- otherwise we'd have infinite recursion!) In this example, there are no rewrite rules for shoelace_data or unit, so rewriting is complete and the above is the final result given to the planner.

Now we face Al with the problem that the Blues Brothers appear in his shop and want to buy some new shoes, and as the Blues Brothers are, they want to wear the same shoes. And they want to wear them immediately, so they need shoelaces too.

Al needs to know for which shoes currently in the store he has the matching shoelaces (color and size) and where the total number of exactly matching pairs is greater or equal to two. We teach him what to do and he asks his database:

```
al_bundy=> SELECT * FROM shoe_ready WHERE total_avail >= 2;
shoename  |sh_avail|sl_name   |sl_avail|total_avail
----------+--------+----------+--------+-----------
sh1       |       2|sl1       |       5|          2
sh3       |       4|sl7       |       7|          4
(2 rows)
```

Al is a shoe guru and so he knows that only shoes of type sh1 would fit (shoelace sl7 is brown and shoes that need brown shoelaces aren't shoes the Blues Brothers would ever wear).

The output of the parser this time is the parse tree

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE int4ge(shoe_ready.total_avail, 2);
```

The first rule applied will be the one for the `shoe_ready` view and it results in the parse tree

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM shoe rsh, shoelace rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE int4ge(shoe_ready.total_avail, 2);
```

Similarly, the rules for `shoe` and `shoelace` are substituted into the range table of the sub-query, leading to a three-level final query tree:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM (SELECT sh.shoename,
                       sh.sh_avail,
                       sh.slcolor,
                       sh.slminlen,
                       sh.slminlen * un.un_fact AS slminlen_cm,
                       sh.slmaxlen,
                       sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                       sh.slunit
                  FROM shoe_data sh, unit un
                 WHERE sh.slunit = un.un_name) rsh,
               (SELECT s.sl_name,
                       s.sl_avail,
                       s.sl_color,
                       s.sl_len,
                       s.sl_unit,
                       s.sl_len * u.un_fact AS sl_len_cm
                  FROM shoelace_data s, unit u
                 WHERE s.sl_unit = u.un_name) rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE int4ge(shoe_ready.total_avail, 2);
```

It turns out that the planner will collapse this tree into a two-level query tree: the bottommost selects will be “pulled up” into the middle select since there's no need to process them separately. But the middle select will remain separate from the top, because it contains aggregate functions. If we pulled those up it would change the behavior of the topmost select, which we don't want. However, collapsing the query tree is an optimization that the rewrite system doesn't have to concern itself with.

Note

There is currently no recursion stopping mechanism for view rules in the rule system (only for the other kinds of rules). This doesn't hurt much, because the only way to push this into an endless loop (blowing up the backend until it reaches the memory limit) is to create tables and then setup the view rules by hand with CREATE RULE in such a way, that one selects from the other that selects from the one. This could never happen if CREATE VIEW is used because for the first CREATE VIEW, the second relation does not exist and thus the first view cannot select from the second.

16.3.3. View Rules in Non-SELECT Statements

Two details of the parse tree aren't touched in the description of view rules above. These are the command type and the result relation. In fact, view rules don't need this information.

There are only a few differences between a parse tree for a SELECT and one for any other command. Obviously they have another command type and this time the result relation points to the range table entry where the result should go. Everything else is absolutely the same. So having two tables t1 and t2 with attributes a and b, the parse trees for the two statements

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b WHERE t1.a = t2.a;
```

are nearly identical.

- The range tables contain entries for the tables t1 and t2.
- The target lists contain one variable that points to attribute b of the range table entry for table t2.
- The qualification expressions compare the attributes a of both ranges for equality.
- The join trees show a simple join between t1 and t2.

The consequence is, that both parse trees result in similar execution plans. They are both joins over the two tables. For the UPDATE the missing columns from t1 are added to the target list by the planner and the final parse tree will read as

```
UPDATE t1 SET a = t1.a, b = t2.b WHERE t1.a = t2.a;
```

and thus the executor run over the join will produce exactly the same result set as a

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

will do. But there is a little problem in UPDATE. The executor does not care what the results from the join it is doing are meant for. It just produces a result set of rows. The difference that one is a SELECT command and the other is an UPDATE is handled in the caller of the executor. The caller still knows (looking at the parse tree) that this is an UPDATE, and he knows that this result should go into table t1. But which of the rows that are there has to be replaced by the new row?

To resolve this problem, another entry is added to the target list in UPDATE (and also in DELETE) statements: the current tuple ID (CTID). This is a system attribute containing the file block number and position in the block for the row. Knowing the table, the CTID can be used to retrieve the original t1 row to be updated. After adding the CTID to the target list, the query actually looks like

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Now another detail of PostgreSQL enters the stage. At this moment, table rows aren't overwritten and this is why ABORT TRANSACTION is fast. In an UPDATE, the new result row is inserted into the table (after stripping CTID) and in the tuple header of the row that CTID pointed to the cmax and xmax entries are set to the current command counter and current transaction ID. Thus the old row is hidden and after the transaction committed the vacuum cleaner can really move it out.

Knowing all that, we can simply apply view rules in absolutely the same way to any command. There is no difference.

16.3.4. The Power of Views in PostgreSQL

The above demonstrates how the rule system incorporates view definitions into the original parse tree. In the second example a simple SELECT from one view created a final parse tree that is a join of 4 tables (unit is used twice with different names).

16.3.4.1. Benefits

The benefit of implementing views with the rule system is, that the planner has all the information about which tables have to be scanned plus the relationships between these tables plus the restrictive qualifications from the views plus the qualifications from the original query in one single parse tree. And this is still the situation when the original query is already a join over views. Now the planner has to decide which is the best path to execute the query. The more information the planner has, the better this decision can be. And the rule system as implemented in PostgreSQL ensures, that this is all information available about the query up to now.

16.3.5. What about updating a view?

What happens if a view is named as the target relation for an INSERT, UPDATE, or DELETE? After doing the substitutions described above, we will have a query tree in which the result relation points at a subquery range table entry. This will not work, so the rewriter throws an error if it sees it has produced such a thing.

To change this we can define rules that modify the behavior of non-SELECT queries. This is the topic of the next section.

16.4. Rules on INSERT, UPDATE and DELETE

16.4.1. Differences from View Rules

Rules that are defined ON INSERT, UPDATE and DELETE are totally different from the view rules described in the previous section. First, their CREATE RULE command allows more:

- They can have no action.
- They can have multiple actions.
- The keyword INSTEAD is optional.
- The pseudo relations NEW and OLD become useful.
- They can have rule qualifications.

Second, they don't modify the parse tree in place. Instead they create zero or many new parse trees and can throw away the original one.

16.4.2. How These Rules Work

Keep the syntax

```
CREATE RULE rule_name AS ON event
    TO object [WHERE rule_qualification]
    DO [INSTEAD] [action | (actions) | NOTHING];
```

in mind. In the following, *update rules* means rules that are defined ON INSERT, UPDATE or DELETE.

Update rules get applied by the rule system when the result relation and the command type of a parse tree are equal to the object and event given in the CREATE RULE command. For update rules, the rule system creates a list of parse trees. Initially the parse tree list is empty. There can be zero (NOTHING keyword), one or multiple actions. To simplify, we look at a rule with one action. This rule can have a qualification or not and it can be INSTEAD or not.

What is a rule qualification? It is a restriction that tells when the actions of the rule should be done and when not. This qualification can only reference the NEW and/or OLD pseudo relations which are basically the relation given as object (but with a special meaning).

So we have four cases that produce the following parse trees for a one-action rule.

- No qualification and not INSTEAD:
  - The parse tree from the rule action where the original parse tree's qualification has been added.
- No qualification but INSTEAD:
  - The parse tree from the rule action where the original parse tree's qualification has been added.
- Qualification given and not INSTEAD:
  - The parse tree from the rule action where the rule qualification and the original parse tree's qualification have been added.
- Qualification given and INSTEAD:
  - The parse tree from the rule action where the rule qualification and the original parse tree's qualification have been added.
  - The original parse tree where the negated rule qualification has been added.

Finally, if the rule is not INSTEAD, the unchanged original parse tree is added to the list. Since only qualified INSTEAD rules already add the original parse tree, we end up with either one or two output parse trees for a rule with one action.

For ON INSERT rules, the original query (if not suppressed by INSTEAD) is done before any actions added by rules. This allows the actions to see the inserted row(s). But for ON UPDATE and ON DELETE rules, the original query is done after the actions added by rules. This ensures that the actions can see the to-be-updated or to-be-deleted rows; otherwise, the actions might do nothing because they find no rows matching their qualifications.

The parse trees generated from rule actions are thrown into the rewrite system again and maybe more rules get applied resulting in more or less parse trees. So the parse trees in the rule actions must have either another command type or another result relation. Otherwise this recursive process will end up in a loop. There is a compiled in recursion limit of currently 10 iterations. If after 10 iterations there are still update rules to apply the rule system assumes a loop over multiple rule definitions and reports an error.

The parse trees found in the actions of the `pg_rewrite` system catalog are only templates. Since they can reference the range-table entries for NEW and OLD, some substitutions have to be made before they can be used. For any reference to NEW, the target list of the original query is searched for a corresponding entry. If found, that entry's expression replaces the reference. Otherwise NEW means the same as OLD (for an UPDATE) or is replaced by NULL (for an INSERT). Any reference to OLD is replaced by a reference to the range-table entry which is the result relation.

After we are done applying update rules, we apply view rules to the produced parse tree(s). Views cannot insert new update actions so there is no need to apply update rules to the output of view rewriting.

16.4.2.1. A First Rule Step by Step

We want to trace changes to the sl_avail column in the `shoelace_data` relation. So we setup a log table and a rule that conditionally writes a log entry when an UPDATE is performed on `shoelace_data`.

```
CREATE TABLE shoelace_log (
    sl_name    char(10),      -- shoelace changed
    sl_avail   integer,       -- new available value
    log_who    text,          -- who did it
    log_when   timestamp      -- when
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail != OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

Now Al does

```
al_bundy=> UPDATE shoelace_data SET sl_avail = 6

al_bundy->     WHERE sl_name = 'sl7';
```

and we look at the log table.

```
al_bundy=> SELECT * FROM shoelace_log;
sl_name   |sl_avail|log_who|log_when
----------+--------+-------+--------------------------------
sl7       |       6|Al     |Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

That's what we expected. What happened in the background is the following. The parser created the parse tree (this time the parts of the original parse tree are highlighted because the base of operations is the rule action for update rules).

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE bpchareq(shoelace_data.sl_name, 'sl7');
```

There is a rule `log_shoelace` that is ON UPDATE with the rule qualification expression

```
int4ne(NEW.sl_avail, OLD.sl_avail)
```

and one action

```
INSERT INTO shoelace_log VALUES(
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp
  FROM shoelace_data *NEW*, shoelace_data *OLD*;
```

This is a little strange-looking since you can't normally write INSERT ... VALUES ... FROM. The FROM clause here is just to indicate that there are range-table entries in the parse tree for *NEW* and *OLD*. These are needed so that they can be referenced by variables in the INSERT command's querytree.

The rule is a qualified non-INSTEAD rule, so the rule system has to return two parse trees: the modified rule action and the original parse tree. In the first step the range table of the original query is incorporated into the rule's action parse tree. This results in

```
INSERT INTO shoelace_log VALUES(
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data;
```

In step 2 the rule qualification is added to it, so the result set is restricted to rows where sl_avail changes.

```
INSERT INTO shoelace_log VALUES(
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE int4ne(*NEW*.sl_avail, *OLD*.sl_avail);
```

This is even stranger-looking, since INSERT ... VALUES doesn't have a WHERE clause either, but the planner and executor will have no difficulty with it. They need to support this same functionality anyway for INSERT ... SELECT. In step 3 the original parse tree's qualification is added, restricting the result set further to only the rows touched by the original parse tree.

```
INSERT INTO shoelace_log VALUES(
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE int4ne(*NEW*.sl_avail, *OLD*.sl_avail)
   AND bpchareq(shoelace_data.sl_name, 'sl7');
```

Step 4 substitutes NEW references by the target list entries from the original parse tree or with the matching variable references from the result relation.

```
INSERT INTO shoelace_log VALUES(
       shoelace_data.sl_name, 6,
       current_user, current_timestamp
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE int4ne(6, *OLD*.sl_avail)
   AND bpchareq(shoelace_data.sl_name, 'sl7');
```

Step 5 changes OLD references into result relation references.

```
INSERT INTO shoelace_log VALUES(
       shoelace_data.sl_name, 6,
       current_user, current_timestamp
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE int4ne(6, shoelace_data.sl_avail)
   AND bpchareq(shoelace_data.sl_name, 'sl7');
```

That's it. Since the rule is not INSTEAD, we also output the original parse tree. In short, the output from the rule system is a list of two parse trees that are the same as the statements:

```
INSERT INTO shoelace_log VALUES(
       shoelace_data.sl_name, 6,
       current_user, current_timestamp
```

```
  FROM shoelace_data
 WHERE 6 != shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

These are executed in this order and that is exactly what the rule defines. The substitutions and the qualifications added ensure that if the original query would be, say,

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

no log entry would get written. This time the original parse tree does not contain a target list entry for sl_avail, so NEW.sl_avail will get replaced by shoelace_data.sl_avail resulting in the extra query

```
INSERT INTO shoelace_log VALUES(
       shoelace_data.sl_name, shoelace_data.sl_avail,
       current_user, current_timestamp)
  FROM shoelace_data
 WHERE shoelace_data.sl_avail != shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

and that qualification will never be true. It will also work if the original query modifies multiple rows. So if Al would issue the command

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

four rows in fact get updated (sl1, sl2, sl3 and sl4). But sl3 already has sl_avail = 0. This time, the original parse trees qualification is different and that results in the extra parse tree

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, 0,
       current_user, current_timestamp
  FROM shoelace_data
 WHERE 0 != shoelace_data.sl_avail
   AND shoelace_data.sl_color = 'black';
```

This parse tree will surely insert three new log entries. And that's absolutely correct.

Here we can see why it is important that the original parse tree is executed last. If the UPDATE would have been executed first, all the rows are already set to zero, so the logging INSERT would not find any row where 0 != shoelace_data.sl_avail.

16.4.3. Cooperation with Views

A simple way to protect view relations from the mentioned possibility that someone can try to INSERT, UPDATE and DELETE on them is to let those parse trees get thrown away. We create the rules

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
```

```
    DO INSTEAD NOTHING;
```

If Al now tries to do any of these operations on the view relation `shoe`, the rule system will apply the rules. Since the rules have no actions and are INSTEAD, the resulting list of parse trees will be empty and the whole query will become nothing because there is nothing left to be optimized or executed after the rule system is done with it.

Note

This way might irritate frontend applications because absolutely nothing happened on the database and thus, the backend will not return anything for the query. Not even a `PGRES_EMPTY_QUERY` will be available in libpq. In psql, nothing happens. This might change in the future.

A more sophisticated way to use the rule system is to create rules that rewrite the parse tree into one that does the right operation on the real tables. To do that on the `shoelace` view, we create the following rules:

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit);

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
    DO INSTEAD
    UPDATE shoelace_data SET
           sl_name = NEW.sl_name,
           sl_avail = NEW.sl_avail,
           sl_color = NEW.sl_color,
           sl_len = NEW.sl_len,
           sl_unit = NEW.sl_unit
     WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

Now there is a pack of shoelaces arriving in Al's shop and it has a big part list. Al is not that good in calculating and so we don't want him to manually update the shoelace view. Instead we setup two little tables, one where he can insert the items from the part list and one with a special trick. The create commands for these are:

```
CREATE TABLE shoelace_arrive (
    arr_name    char(10),
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     char(10),
    ok_quant    integer
);
```

```
CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace SET
           sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

Now Al can sit down and do whatever until

```
al_bundy=> SELECT * FROM shoelace_arrive;
arr_name  |arr_quant
----------+---------
sl3       |       10
sl6       |       20
sl8       |       20
(3 rows)
```

is exactly what's on the part list. We take a quick look at the current data,

```
al_bundy=> SELECT * FROM shoelace;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       6|brown     |    60|cm      |       60
sl3       |       0|black     |    35|inch    |     88.9
sl4       |       8|black     |    40|inch    |    101.6
sl8       |       1|brown     |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |       0|brown     |   0.9|m       |       90
(8 rows)
```

move the arrived shoelaces in

```
al_bundy=> INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

and check the results

```
al_bundy=> SELECT * FROM shoelace ORDER BY sl_name;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       6|brown     |    60|cm      |       60
sl4       |       8|black     |    40|inch    |    101.6
sl3       |      10|black     |    35|inch    |     88.9
sl8       |      21|brown     |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |      20|brown     |   0.9|m       |       90
(8 rows)

al_bundy=> SELECT * FROM shoelace_log;
sl_name   |sl_avail|log_who|log_when
----------+--------+-------+--------------------------------
sl7       |       6|Al     |Tue Oct 20 19:14:45 1998 MET DST
```

```
sl3       |      10|Al     |Tue Oct 20 19:25:16 1998 MET DST
sl6       |      20|Al     |Tue Oct 20 19:25:16 1998 MET DST
sl8       |      21|Al     |Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

It's a long way from the one INSERT ... SELECT to these results. And its description will be the last in this document (but not the last example :-). First there was the parser's output

```
INSERT INTO shoelace_ok SELECT
       shoelace_arrive.arr_name, shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

Now the first rule `shoelace_ok_ins` is applied and turns it into

```
UPDATE shoelace SET
       sl_avail = int4pl(shoelace.sl_avail, shoelace_arrive.arr_quant)
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace
 WHERE bpchareq(shoelace.sl_name, showlace_arrive.arr_name);
```

and throws away the original INSERT on `shoelace_ok`. This rewritten query is passed to the rule system again and the second applied rule `shoelace_upd` produced

```
UPDATE shoelace_data SET
       sl_name = shoelace.sl_name,
       sl_avail = int4pl(shoelace.sl_avail,
 shoelace_arrive.arr_quant),
       sl_color = shoelace.sl_color,
       sl_len = shoelace.sl_len,
       sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data showlace_data
 WHERE bpchareq(shoelace.sl_name, showlace_arrive.arr_name)
   AND bpchareq(shoelace_data.sl_name, shoelace.sl_name);
```

Again it's an INSTEAD rule and the previous parse tree is trashed. Note that this query still uses the view `shoelace`. But the rule system isn't finished with this loop so it continues and applies the rule `_RETshoelace` on it and we get

```
UPDATE shoelace_data SET
       sl_name = s.sl_name,
       sl_avail = int4pl(s.sl_avail, shoelace_arrive.arr_quant),
       sl_color = s.sl_color,
       sl_len = s.sl_len,
       sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data showlace_data,
       shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u
 WHERE bpchareq(s.sl_name, showlace_arrive.arr_name)
   AND bpchareq(shoelace_data.sl_name, s.sl_name);
```

Again an update rule has been applied and so the wheel turns on and we are in rewrite round 3. This time rule `log_shoelace` gets applied what produces the extra parse tree

```
INSERT INTO shoelace_log SELECT
       s.sl_name,
       int4pl(s.sl_avail, shoelace_arrive.arr_quant),
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data showlace_data,
       shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u,
       shoelace_data *OLD*, shoelace_data *NEW*
       shoelace_log shoelace_log
 WHERE bpchareq(s.sl_name,  showlace_arrive.arr_name)
   AND bpchareq(shoelace_data.sl_name, s.sl_name);
   AND int4ne(int4pl(s.sl_avail, shoelace_arrive.arr_quant),
 s.sl_avail);
```

After that the rule system runs out of rules and returns the generated parse trees. So we end up with two final parse trees that are equal to the SQL statements

```
INSERT INTO shoelace_log SELECT
       s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND s.sl_avail + shoelace_arrive.arr_quant != s.sl_avail;

UPDATE shoelace_data SET
       sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive,
       shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.sl_name
   AND shoelace_data.sl_name = s.sl_name;
```

The result is that data coming from one relation inserted into another, changed into updates on a third, changed into updating a fourth plus logging that final update in a fifth gets reduced into two queries.

There is a little detail that's a bit ugly. Looking at the two queries turns out, that the `shoelace_data` relation appears twice in the range table where it could definitely be reduced to one. The planner does not handle it and so the execution plan for the rule systems output of the INSERT will be

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
```

```
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

while omitting the extra range table entry would result in a

```
Merge Join
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on shoelace_arrive
```

that totally produces the same entries in the log relation. Thus, the rule system caused one extra scan on the `shoelace_data` relation that is absolutely not necessary. And the same obsolete scan is done once more in the UPDATE. But it was a really hard job to make that all possible at all.

A final demonstration of the PostgreSQL rule system and its power. There is a cute blonde that sells shoelaces. And what Al could never realize, she's not only cute, she's smart too - a little too smart. Thus, it happens from time to time that Al orders shoelaces that are absolutely not sellable. This time he ordered 1000 pairs of magenta shoelaces and since another kind is currently not available but he committed to buy some, he also prepared his database for pink ones.

```
al_bundy=> INSERT INTO shoelace VALUES
al_bundy->     ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
al_bundy=> INSERT INTO shoelace VALUES
al_bundy->     ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

Since this happens often, we must lookup for shoelace entries, that fit for absolutely no shoe sometimes. We could do that in a complicated statement every time, or we can setup a view for it. The view for this is

```
CREATE VIEW shoelace_obsolete AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

Its output is

```
al_bundy=> SELECT * FROM shoelace_obsolete;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl9       |       0|pink      |    35|inch    |     88.9
sl10      |    1000|magenta   |    40|inch    |    101.6
```

For the 1000 magenta shoelaces we must debt Al before we can throw 'em away, but that's another problem. The pink entry we delete. To make it a little harder for PostgreSQL, we don't delete it directly. Instead we create one more view

```
CREATE VIEW shoelace_candelete AS
    SELECT * FROM shoelace_obsolete WHERE sl_avail = 0;
```

and do it this way:

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_candelete
             WHERE sl_name = shoelace.sl_name);
```

Voilà:

```
al_bundy=> SELECT * FROM shoelace;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       6|brown     |    60|cm      |       60
sl4       |       8|black     |    40|inch    |    101.6
sl3       |      10|black     |    35|inch    |     88.9
sl8       |      21|brown     |    40|inch    |    101.6
sl10      |    1000|magenta   |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |      20|brown     |   0.9|m       |       90
(9 rows)
```

A DELETE on a view, with a subselect qualification that in total uses 4 nesting/joined views, where one of them itself has a subselect qualification containing a view and where calculated view columns are used, gets rewritten into one single parse tree that deletes the requested data from a real table.

I think there are only a few situations out in the real world, where such a construct is necessary. But it makes me feel comfortable that it works.

The truth is: Doing this I found one more bug while writing this document. But after fixing that I was a little amazed that it works at all.

16.5. Rules and Permissions

Due to rewriting of queries by the PostgreSQL rule system, other tables/views than those used in the original query get accessed. Using update rules, this can include write access to tables.

Rewrite rules don't have a separate owner. The owner of a relation (table or view) is automatically the owner of the rewrite rules that are defined for it. The PostgreSQL rule system changes the behavior of the default access control system. Relations that are used due to rules get checked against the permissions of the rule owner, not the user invoking the rule. This means, that a user does only need the required permissions for the tables/views he names in his queries.

For example: A user has a list of phone numbers where some of them are private, the others are of interest for the secretary of the office. He can construct the following:

```
CREATE TABLE phone_data (person text, phone text, private bool);
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE NOT private;
GRANT SELECT ON phone_number TO secretary;
```

Nobody except him (and the database superusers) can access the phone_data table. But due to the GRANT, the secretary can SELECT from the phone_number view. The rule system will rewrite the SELECT from phone_number into a SELECT from phone_data and add the qualification that only entries where private is false are wanted. Since the user is the owner of phone_number, the read access to phone_data is now checked against his permissions and the query is considered granted. The check for accessing phone_number is also performed, but this is done against the invoking user, so nobody but the user and the secretary can use it.

The permissions are checked rule by rule. So the secretary is for now the only one who can see the public phone numbers. But the secretary can setup another view and grant access to that to public. Then, anyone

can see the phone_number data through the secretaries view. What the secretary cannot do is to create a view that directly accesses phone_data (actually he can, but it will not work since every access aborts the transaction during the permission checks). And as soon as the user will notice, that the secretary opened his phone_number view, he can REVOKE his access. Immediately any access to the secretaries view will fail.

Someone might think that this rule by rule checking is a security hole, but in fact it isn't. If this would not work, the secretary could setup a table with the same columns as phone_number and copy the data to there once per day. Then it's his own data and he can grant access to everyone he wants. A GRANT means “I trust you”. If someone you trust does the thing above, it's time to think it over and then REVOKE.

This mechanism does also work for update rules. In the examples of the previous section, the owner of the tables in Al's database could GRANT SELECT, INSERT, UPDATE and DELETE on the shoelace view to al. But only SELECT on shoelace_log. The rule action to write log entries will still be executed successfully. And Al could see the log entries. But he cannot create fake entries, nor could he manipulate or remove existing ones.

Warning

GRANT ALL currently includes RULE permission. This means the granted user could drop the rule, do the changes and reinstall it. I think this should get changed quickly.

16.6. Rules versus Triggers

Many things that can be done using triggers can also be implemented using the PostgreSQL rule system. What currently cannot be implemented by rules are some kinds of constraints. It is possible, to place a qualified rule that rewrites a query to NOTHING if the value of a column does not appear in another table. But then the data is silently thrown away and that's not a good idea. If checks for valid values are required, and in the case of an invalid value an error message should be generated, it must be done by a trigger for now.

On the other hand a trigger that is fired on INSERT on a view can do the same as a rule, put the data somewhere else and suppress the insert in the view. But it cannot do the same thing on UPDATE or DELETE, because there is no real data in the view relation that could be scanned and thus the trigger would never get called. Only a rule will help.

For the things that can be implemented by both, it depends on the usage of the database, which is the best. A trigger is fired for any row affected once. A rule manipulates the parse tree or generates an additional one. So if many rows are affected in one statement, a rule issuing one extra query would usually do a better job than a trigger that is called for any single row and must execute his operations this many times.

For example: There are two tables

```
CREATE TABLE computer (
    hostname        text,    -- indexed
    manufacturer    text     -- indexed
);

CREATE TABLE software (
    software        text,    -- indexed
    hostname        text     -- indexed
);
```

Both tables have many thousands of rows and the index on `hostname` is unique. The `hostname` column contains the full qualified domain name of the computer. The rule/trigger should constraint delete rows

from software that reference the deleted host. Since the trigger is called for each individual row deleted from computer, it can use the statement

```
DELETE FROM software WHERE hostname = $1;
```

in a prepared and saved plan and pass the `hostname` in the parameter. The rule would be written as

```
CREATE RULE computer_del AS ON DELETE TO computer
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

Now we look at different types of deletes. In the case of a

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

the table computer is scanned by index (fast) and the query issued by the trigger would also be an index scan (fast too). The extra query from the rule would be a

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                       AND software.hostname = computer.hostname;
```

Since there are appropriate indexes setup, the planner will create a plan of

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

So there would be not that much difference in speed between the trigger and the rule implementation. With the next delete we want to get rid of all the 2000 computers where the `hostname` starts with 'old'. There are two possible queries to do that. One is

```
DELETE FROM computer WHERE hostname >= 'old'
                       AND hostname <  'ole'
```

Where the plan for the rule query will be a

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
    ->  Index Scan using comp_hostidx on computer
```

The other possible query is a

```
DELETE FROM computer WHERE hostname ~ '^old';
```

with the execution plan

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

This shows, that the planner does not realize that the qualification for the `hostname` on computer could also be used for an index scan on software when there are multiple qualification expressions combined with AND, what he does in the regexp version of the query. The trigger will get invoked once for any of the 2000 old computers that have to be deleted and that will result in one index scan over computer and 2000 index scans for the software. The rule implementation will do it with two queries over indexes. And it depends on the overall size of the software table if the rule will still be faster in the sequential scan

situation. 2000 query executions over the SPI manager take some time, even if all the index blocks to look them up will soon appear in the cache.

The last query we look at is a

```
DELETE FROM computer WHERE manufacurer = 'bim';
```

Again this could result in many rows to be deleted from computer. So the trigger will again fire many queries into the executor. But the rule plan will again be the nested loop over two index scans. Only using another index on computer:

```
Nestloop
  ->  Index Scan using comp_manufidx on computer
  ->  Index Scan using soft_hostidx on software
```

resulting from the rules query

```
DELETE FROM software WHERE computer.manufacurer = 'bim'
                       AND software.hostname = computer.hostname;
```

In any of these cases, the extra queries from the rule system will be more or less independent from the number of affected rows in a query.

Another situation is cases on UPDATE where it depends on the change of an attribute if an action should be performed or not. In PostgreSQL version 6.4, the attribute specification for rule events is disabled (it will have its comeback latest in 6.5, maybe earlier - stay tuned). So for now the only way to create a rule as in the shoelace_log example is to do it with a rule qualification. That results in an extra query that is performed always, even if the attribute of interest cannot change at all because it does not appear in the target list of the initial query. When this is enabled again, it will be one more advantage of rules over triggers. Optimization of a trigger must fail by definition in this case, because the fact that its actions will only be done when a specific attribute is updated is hidden in its functionality. The definition of a trigger only allows to specify it on row level, so whenever a row is touched, the trigger must be called to make its decision. The rule system will know it by looking up the target list and will suppress the additional query completely if the attribute isn't touched. So the rule, qualified or not, will only do its scans if there ever could be something to do.

Rules will only be significantly slower than triggers if their actions result in large and bad qualified joins, a situation where the planner fails. They are a big hammer. Using a big hammer without caution can cause big damage. But used with the right touch, they can hit any nail on the head.

Chapter 17. Interfacing Extensions To Indexes

17.1. Introduction

The procedures described thus far let you define new types, new functions, and new operators. However, we cannot yet define a secondary index (such as a B-tree, R-tree, or hash access method) over a new type or its operators.

Look back at Figure 11.1, "The major PostgreSQL system catalogs". The right half shows the catalogs that we must modify in order to tell PostgreSQL how to use a user-defined type and/or user-defined operators with an index (i.e., `pg_am`, `pg_amop`, `pg_amproc`, `pg_operator` and `pg_opclass`). Unfortunately, there is no simple command to do this. We will demonstrate how to modify these catalogs through a running example: a new operator class for the B-tree access method that stores and sorts complex numbers in ascending absolute value order.

17.2. Access Methods

The `pg_am` table contains one row for every index access method. Support for the heap access method is built into PostgreSQL, but all other access methods are described in `pg_am`. The schema is shown in Table 17.1, "Index Access Method Schema".

Table 17.1. Index Access Method Schema

Column	Description
amname	name of the access method
amowner	user ID of the owner (currently not used)
amstrategies	number of strategies for this access method (see below)
amsupport	number of support routines for this access method (see below)
amorderstrategy	zero if the index offers no sort order, otherwise the strategy number of the strategy operator that describes the sort order
amcanunique	does AM support unique indexes?
amcanmulticol	does AM support multicolumn indexes?
amindexnulls	does AM support NULL index entries?
amconcurrent	does AM support concurrent updates?
amgettuple	
aminsert	
...	procedure identifiers for interface routines to the access method. For example, regproc IDs for opening, closing, and getting rows from the access method appear here.

The object ID of the row in pg_am is used as a foreign key in a lot of other tables. You do not need to add a new row to this table; all that you are interested in is the object ID of the access method you want to extend:

```
SELECT oid FROM pg_am WHERE amname = 'btree';

 oid
-----
 403
(1 row)
```

We will use that query in a WHERE clause later.

17.3. Access Method Strategies

The amstrategies column exists to standardize comparisons across data types. For example, B-trees impose a strict ordering on keys, lesser to greater. Since 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. In fact, some access methods don't impose any ordering at all. For example, R-trees express a rectangle-containment relationship, whereas a hashed data structure expresses only bitwise similarity based on the value of a hash function. PostgreSQL needs some consistent way of taking a qualification in your query, looking at the operator, and then deciding if a usable index exists. This implies that PostgreSQL needs to know, for example, that the <= and > operators partition a B-tree. PostgreSQL uses *strategies* to express these relationships between operators and the way they can be used to scan indexes.

Defining a new set of strategies is beyond the scope of this discussion, but we'll explain how B-tree strategies work because you'll need to know that to add a new B-tree operator class. In the pg_am table, the amstrategies column sets the number of strategies defined for this access method. For B-trees, this number is 5. The meanings of these strategies are shown in Table 17.2, “B-tree”.

Table 17.2. B-tree Strategies

Operation	Index
less than	1
less than or equal	2
equal	3
greater than or equal	4
greater than	5

The idea is that you'll need to add operators corresponding to these strategies to the pg_amop relation (see below). The access method code can use these strategy numbers, regardless of data type, to figure out how to partition the B-tree, compute selectivity, and so on. Don't worry about the details of adding operators yet; just understand that there must be a set of these operators for int2, int4, oid, and all other data types on which a B-tree can operate.

17.4. Access Method Support Routines

Sometimes, strategies aren't enough information for the system to figure out how to use an index. Some access methods require additional support routines in order to work. For example, the B-tree access 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 access method must be able to compute intersections, unions, and sizes of rectangles. These operations do not correspond to operators used in qualifications in SQL queries; they are administrative routines used by the access methods, internally.

In order to manage diverse support routines consistently across all PostgreSQL access methods, pg_am includes a column called amsupport. This column records the number of support routines used by an access method. For B-trees, this number is one: the routine to take two keys and return -1, 0, or +1, depending on whether the first key is less than, equal to, or greater than the second. (Strictly speaking, this routine can return a negative number (< 0), zero, or a non-zero positive number (> 0).)

The amstrategies entry in pg_am is just the number of strategies defined for the access method in question. The operators for less than, less equal, and so on don't appear in pg_am. Similarly, amsupport is just the number of support routines required by the access method. The actual routines are listed elsewhere.

By the way, the amorderstrategy column tells whether the access method supports ordered scan. Zero means it doesn't; if it does, amorderstrategy is the number of the strategy routine that corresponds to the ordering operator. For example, B-tree has amorderstrategy = 1, which is its “less than” strategy number.

17.5. Operator Classes

The next table of interest is pg_opclass. This table defines operator class names and input data types for each of the operator classes supported by a given index access method. The same class name can be used for several different access methods (for example, both B-tree and hash access methods have operator classes named oid_ops), but a separate pg_opclass row must appear for each access method. The OID of the pg_opclass row is used as a foreign key in other tables to associate specific operators and support routines with the operator class.

You need to add a row with your operator class name (for example, complex_abs_ops) to pg_opclass:

```
INSERT INTO pg_opclass (opcamid, opcname, opcintype, opcdefault,
 opckeytype)
    VALUES (
         (SELECT oid FROM pg_am WHERE amname = 'btree'),
         'complex_abs_ops',
         (SELECT oid FROM pg_type WHERE typname = 'complex'),
         true,
         0);

SELECT oid, *
    FROM pg_opclass
    WHERE opcname = 'complex_abs_ops';

  oid   | opcamid |     opcname     | opcintype | opcdefault |
 opckeytype
--------+---------+-----------------+-----------+------------
+------------
 277975 |     403 | complex_abs_ops |    277946 | t          |
   0
(1 row)
```

Note that the OID for your pg_opclass row will be different! Don't worry about this though. We'll get this number from the system later just like we got the OID of the type here.

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 set opcdefault to false instead. opckeytype is not described here; it should always be zero for B-tree operator classes.

17.6. Creating the Operators and Support Routines

So now we have an access method and an operator class. We still need a set of operators. The procedure for defining operators was discussed in Chapter 14, *Extending SQL: Operators*. For the `complex_abs_ops` 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)

Suppose the code that implements these functions is stored in the file *PGROOT*`/src/tutorial/complex.c`, which we have compiled into *PGROOT*`/src/tutorial/complex.so`. Part of the C code looks like this:

```
#define Mag(c) ((c)->x*(c)->x + (c)->y*(c)->y)

        bool
        complex_abs_eq(Complex *a, Complex *b)
        {
            double amag = Mag(a), bmag = Mag(b);
            return (amag==bmag);
        }
```

(Note that we will only show the equality operator for the rest of the examples. The other four operators are very similar. Refer to `complex.c` or `complex.source` for the details.)

We make the function known to PostgreSQL like this:

```
CREATE FUNCTION complex_abs_eq(complex, complex) RETURNS boolean
    AS 'PGROOT/src/tutorial/complex'
    LANGUAGE C;
```

There are some important things that are happening here:

- First, note that operators for less-than, less-than-or-equal, equal, greater-than-or-equal, and greater-than for `complex` are being defined. We can only have 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. In that case, we'd need to use some other operator name for `complex_abs_eq`.

- Second, although PostgreSQL can cope with operators having the same name as long as they have different input data types, C can only cope with one global routine having a given name, period. 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.

- Third, we could have made the PostgreSQL name of the function `abs_eq`, relying on PostgreSQL to distinguish it by input 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.

- Finally, note that these operator functions return Boolean values. In practice, all operators defined as index access method strategies must return type `boolean`, since they must appear at the top level of a

WHERE clause to be used with an index. (On the other hand, the support function returns whatever the particular access method expects -- in this case, a signed integer.)

The final routine in the file is the "support routine" mentioned when we discussed the amsupport column of the pg_am table. We will use this later on. For now, ignore it.

Now we are ready to define the operators:

```
CREATE OPERATOR = (
     leftarg = complex, rightarg = complex,
     procedure = complex_abs_eq,
     restrict = eqsel, join = eqjoinsel
         );
```

The important things here are the procedure names (which are the C functions defined above) and the restriction and join selectivity functions. You should just use the selectivity functions used in the example (see complex.source). Note that there are different such functions for the less-than, equal, and greater-than cases. These must be supplied or the optimizer will be unable to make effective use of the index.

The next step is to add entries for these operators to the pg_amop relation. To do this, we'll need the OIDs of the operators we just defined. We'll look up the names of all the operators that take two operands of type complex, and pick ours out:

```
SELECT o.oid AS opoid, o.oprname
    INTO TEMP TABLE complex_ops_tmp
    FROM pg_operator o, pg_type t
    WHERE o.oprleft = t.oid and o.oprright = t.oid
      and t.typname = 'complex';

 opoid  | oprname
--------+---------
 277963 | +
 277970 | <
 277971 | <=
 277972 | =
 277973 | >=
 277974 | >
(6 rows)
```

(Again, some of your OID numbers will almost certainly be different.) The operators we are interested in are those with OIDs 277970 through 277974. The values you get will probably be different, and you should substitute them for the values below. We will do this with a select statement.

Now we are ready to insert entries into pg_amop for our new operator class. These entries must associate the correct B-tree strategy numbers with each of the operators we need. The command to insert the less-than operator looks like:

```
INSERT INTO pg_amop (amopclaid, amopstrategy, amopreqcheck, amopopr)
    SELECT opcl.oid, 1, false, c.opoid
        FROM pg_opclass opcl, complex_ops_tmp c
        WHERE
            opcamid = (SELECT oid FROM pg_am WHERE amname = 'btree')
 AND
            opcname = 'complex_abs_ops' AND
            c.oprname = '<';
```

Now do this for the other operators substituting for the `1` in the second line above and the < in the last line. Note the order: “less than” is 1, “less than or equal” is 2, “equal” is 3, “greater than or equal” is 4, and “greater than” is 5.

The field `amopreqcheck` is not discussed here; it should always be false for B-tree operators.

The final step is the registration of the “support routine” previously described in our discussion of `pg_am`. The OID of this support routine is stored in the `pg_amproc` table, keyed by the operator class OID and the support routine number.

First, we need to register the function in PostgreSQL (recall that we put the C code that implements this routine in the bottom of the file in which we implemented the operator routines):

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'PGROOT/src/tutorial/complex'
    LANGUAGE C;

SELECT oid, proname FROM pg_proc
    WHERE proname = 'complex_abs_cmp';

  oid   |     proname
--------+-----------------
 277997 | complex_abs_cmp
(1 row)
```

(Again, your OID number will probably be different.)

We can add the new row as follows:

```
INSERT INTO pg_amproc (amopclaid, amprocnum, amproc)
    SELECT opcl.oid, 1, p.oid
        FROM pg_opclass opcl, pg_proc p
        WHERE
            opcamid = (SELECT oid FROM pg_am WHERE amname = 'btree')
 AND
            opcname = 'complex_abs_ops' AND
            p.proname = 'complex_abs_cmp';
```

And we're done! (Whew.) It should now be possible to create and use B-tree indexes on `complex` columns.

Chapter 18. Index Cost Estimation Functions

Author

Written by Tom Lane (<tgl@sss.pgh.pa.us>) on 2000-01-24

Note

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.

Note

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 tuples 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 tuple 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 tuples 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 tuple. 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 tuples that will be retrieved during the index scan. In the case of a lossy index, this will typically be higher than the fraction of tuples 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 tuples from the main table.

Cost Estimation

A typical cost estimator will proceed as follows:

1. Estimate and return the fraction of main-table tuples 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,
                                           lfirsti(rel->relids));
```

2. Estimate the number of index tuples that will be visited during the scan. For many index types this is the same as indexSelectivity times the number of tuples in the index, but it might be more. (Note that the index's size in pages and tuples 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 tuple.
     * All the costs are assumed to be paid incrementally during
the scan.
     */
    *indexStartupCost = 0;
    *indexTotalCost = numIndexPages +
        (cpu_index_tuple_cost + cost_qual_eval(indexQuals)) *
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/adt/selfuncs.c`.

By convention, the `pg_proc` entry for an `amcostestimate` function should show

```
prorettype = 0
pronargs = 8
proargtypes = 0 0 0 0 0 0 0 0
```

We use zero ("opaque") for all the arguments since none of them have types that are known in pg_type.

Chapter 19. GiST Indexes

Gene Selkov

The information about GIST is at http://GiST.CS.Berkeley.EDU:8000/gist/ with more on different indexing and sorting schemes at http://s2k-ftp.CS.Berkeley.EDU:8000/personal/jmh/. And there is more interesting reading at http://epoch.cs.berkeley.edu:8000/ and http://www.sai.msu.su/~megera/postgres/gist/.

Author

This extraction from an email sent by Eugene Selkov, Jr. (<selkovjr@mcs.anl.gov>) contains good information on GiST. Hopefully we will learn more in the future and update this information. - thomas 1998-03-01

Well, I can't say I quite understand what's going on, but at least I (almost) succeeded in porting GiST examples to linux. The GiST access method is already in the postgres tree (`src/backend/access/gist`).

Examples at Berkeley[1] come with an overview of the methods and demonstrate spatial index mechanisms for 2D boxes, polygons, integer intervals and text (see also GiST at Berkeley[2]). In the box example, we are supposed to see a performance gain when using the GiST index; it did work for me but I do not have a reasonably large collection of boxes to check that. Other examples also worked, except polygons: I got an error doing

```
test=> CREATE INDEX pix ON polytmp
test-> USING GIST (p:box gist_poly_ops) WITH (ISLOSSY);
ERROR:  cannot open pix

(PostgreSQL 6.3              Sun Feb  1 14:57:30 EST 1998)
```

I could not get sense of this error message; it appears to be something we'd rather ask the developers about (see also Note 4 below). What I would suggest here is that someone of you linux guys (linux==gcc?) fetch the original sources quoted above and apply my patch (see attachment) and tell us what you feel about it. Looks cool to me, but I would not like to hold it up while there are so many competent people around.

A few notes on the sources:

1. I failed to make use of the original (HP-UX) Makefile and rearranged the Makefile from the ancient postgres95 tutorial to do the job. I tried to keep it generic, but I am a very poor makefile writer -- just did some monkey work. Sorry about that, but I guess it is now a little more portable that the original makefile.

2. I built the example sources right under pgsql/src (just extracted the tar file there). The aforementioned Makefile assumes it is one level below pgsql/src (in our case, in pgsql/src/pggist).

3. The changes I made to the *.c files were all about #include's, function prototypes and typecasting. Other than that, I just threw away a bunch of unused vars and added a couple parentheses to please gcc. I hope I did not screw up too much :)

4. There is a comment in polyproc.sql:

```
-- -- there's a memory leak in rtree poly_ops!!
-- -- CREATE INDEX pix2 ON polytmp USING RTREE (p poly_ops);
```

[1] ftp://s2k-ftp.cs.berkeley.edu/pub/gist/pggist/pggist.tgz
[2] http://gist.cs.berkeley.edu:8000/gist/

Roger that!! I thought it could be related to a number of PostgreSQL versions back and tried the query. My system went nuts and I had to shoot down the postmaster in about ten minutes.

I will continue to look into GiST for a while, but I would also appreciate more examples of R-tree usage.

Chapter 20. Triggers

PostgreSQL has various server-side function interfaces. Server-side functions can be written in SQL, PLPGSQL, TCL, or C. Trigger functions can be written in any of these languages except SQL. Note that STATEMENT-level trigger events are not supported in the current version. You can currently specify BEFORE or AFTER on INSERT, DELETE or UPDATE of a tuple as a trigger event.

20.1. Trigger Creation

If a trigger event occurs, the trigger manager (called by the Executor) sets up a TriggerData information structure (described below) and calls the trigger function to handle the event.

The trigger function must be defined before the trigger is created as a function taking no arguments and returning opaque. If the function is written in C, it must use the "version 1" function manager interface.

The syntax for creating triggers is as follows:

```
CREATE TRIGGER trigger [ BEFORE | AFTER ] [ INSERT | DELETE | UPDATE
 [ OR ... ] ]
    ON relation FOR EACH [ ROW | STATEMENT ]
    EXECUTE PROCEDURE procedure
     (args);
```

where the arguments are:

trigger

The name of the trigger is used if you ever have to delete the trigger. It is used as an argument to the `DROP TRIGGER` command.

BEFORE
AFTER

Determines whether the function is called before or after the event.

INSERT
DELETE
UPDATE

The next element of the command determines on what event(s) will trigger the function. Multiple events can be specified separated by OR.

relation

The relation name determines which table the event applies to.

ROW
STATEMENT

The FOR EACH clause determines whether the trigger is fired for each affected row or before (or after) the entire statement has completed.

procedure

The procedure name is the function called.

args

The arguments passed to the function in the TriggerData structure. The purpose of passing arguments to the function is to allow different triggers with similar requirements to call the same function.

Also, *procedure* may be used for triggering different relations (these functions are named as *general trigger functions*).

As example of using both features above, there could be a general function that takes as its arguments two field names and puts the current user in one and the current timestamp in the other. This allows triggers to be written on INSERT events to automatically track creation of records in a transaction table for example. It could also be used as a "last updated" function if used in an UPDATE event.

Trigger functions return HeapTuple to the calling Executor. This is ignored for triggers fired after an INSERT, DELETE or UPDATE operation but it allows BEFORE triggers to:

- Return NULL to skip the operation for the current tuple (and so the tuple will not be inserted/updated/deleted).
- Return a pointer to another tuple (INSERT and UPDATE only) which will be inserted (as the new version of the updated tuple if UPDATE) instead of original tuple.

Note that there is no initialization performed by the CREATE TRIGGER handler. This will be changed in the future. Also, if more than one trigger is defined for the same event on the same relation, the order of trigger firing is unpredictable. This may be changed in the future.

If a trigger function executes SQL-queries (using SPI) then these queries may fire triggers again. This is known as cascading triggers. There is no explicit limitation on the number of cascade levels.

If a trigger is fired by INSERT and inserts a new tuple in the same relation then this trigger will be fired again. Currently, there is nothing provided for synchronization (etc) of these cases but this may change. At the moment, there is function funny_dup17() in the regress tests which uses some techniques to stop recursion (cascading) on itself...

20.2. Interaction with the Trigger Manager

This section describes the low-level details of the interface to a trigger function. This information is only needed when writing a trigger function in C. If you are using a higher-level function language then these details are handled for you.

Note

The interface described here applies for PostgreSQL 7.1 and later. Earlier versions passed the TriggerData pointer in a global variable CurrentTriggerData.

When a function is called by the trigger manager, it is not passed any normal parameters, but it is passed a "context" pointer pointing to a TriggerData structure. C functions can check whether they were called from the trigger manager or not by executing the macro `CALLED_AS_TRIGGER(fcinfo)`, which expands to

```
        ((fcinfo)->context != NULL && IsA((fcinfo)->context,
 TriggerData))
```

If this returns TRUE, then it is safe to cast fcinfo->context to type `TriggerData *` and make use of the pointed-to TriggerData structure. The function must *not* alter the TriggerData structure or any of the data it points to.

struct TriggerData is defined in src/include/commands/trigger.h:

```
typedef struct TriggerData
{
    NodeTag       type;
    TriggerEvent  tg_event;
    Relation      tg_relation;
    HeapTuple     tg_trigtuple;
    HeapTuple     tg_newtuple;
    Trigger      *tg_trigger;
} TriggerData;
```

where the members are defined as follows:

type

Always `T_TriggerData` if this is a trigger event.

tg_event

describes the event for which the function is called. You may use the following macros to examine `tg_event`:

TRIGGER_FIRED_BEFORE(tg_event)

returns TRUE if trigger fired BEFORE.

TRIGGER_FIRED_AFTER(tg_event)

Returns TRUE if trigger fired AFTER.

TRIGGER_FIRED_FOR_ROW(event)

Returns TRUE if trigger fired for a ROW-level event.

TRIGGER_FIRED_FOR_STATEMENT(event)

Returns TRUE if trigger fired for STATEMENT-level event.

TRIGGER_FIRED_BY_INSERT(event)

Returns TRUE if trigger fired by INSERT.

TRIGGER_FIRED_BY_DELETE(event)

Returns TRUE if trigger fired by DELETE.

TRIGGER_FIRED_BY_UPDATE(event)

Returns TRUE if trigger fired by UPDATE.

tg_relation

is a pointer to structure describing the triggered relation. Look at src/include/utils/rel.h for details about this structure. The most interest things are tg_relation->rd_att (descriptor of the relation

tuples) and tg_relation->rd_rel->relname (relation's name. This is not char*, but NameData. Use SPI_getrelname(tg_relation) to get char* if you need a copy of name).

tg_trigtuple

is a pointer to the tuple for which the trigger is fired. This is the tuple being inserted (if INSERT), deleted (if DELETE) or updated (if UPDATE). If INSERT/DELETE then this is what you are to return to Executor if you don't want to replace tuple with another one (INSERT) or skip the operation.

tg_newtuple

is a pointer to the new version of tuple if UPDATE and NULL if this is for an INSERT or a DELETE. This is what you are to return to Executor if UPDATE and you don't want to replace this tuple with another one or skip the operation.

tg_trigger

is pointer to structure Trigger defined in src/include/utils/rel.h:

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    bool        tgenabled;
    bool        tgisconstraint;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgattr[FUNC_MAX_ARGS];
    char      **tgargs;
} Trigger;
```

where tgname is the trigger's name, tgnargs is number of arguments in tgargs, tgargs is an array of pointers to the arguments specified in the CREATE TRIGGER statement. Other members are for internal use only.

20.3. Visibility of Data Changes

PostgreSQL data changes visibility rule: during a query execution, data changes made by the query itself (via SQL-function, SPI-function, triggers) are invisible to the query scan. For example, in query

```
INSERT INTO a SELECT * FROM a;
```

tuples inserted are invisible for SELECT scan. In effect, this duplicates the database table within itself (subject to unique index rules, of course) without recursing.

But keep in mind this notice about visibility in the SPI documentation:

> Changes made by query Q are visible by queries that are started after query Q, no matter whether they are started inside Q (during the execution of Q) or after Q is done.

This is true for triggers as well so, though a tuple being inserted (tg_trigtuple) is not visible to queries in a BEFORE trigger, this tuple (just inserted) is visible to queries in an AFTER trigger, and to queries in BEFORE/AFTER triggers fired after this!

20.4. Examples

There are more complex examples in `src/test/regress/regress.c` and in `contrib/spi`.

Here is a very simple example of trigger usage. Function trigf reports the number of tuples in the triggered relation ttest and skips the operation if the query attempts to insert NULL into x (i.e - it acts as a NOT NULL constraint but doesn't abort the transaction).

```
#include "executor/spi.h" /* this is what you need to work with SPI */
#include "commands/trigger.h" /* -"- and triggers */

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;

 /* Make sure trigdata is pointing at what I expect */
 if (!CALLED_AS_TRIGGER(fcinfo))
  elog(ERROR, "trigf: not fired by trigger manager");

 /* tuple to return to Executor */
 if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
  rettuple = trigdata->tg_newtuple;
 else
  rettuple = trigdata->tg_trigtuple;

 /* check for NULLs ? */
 if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event) &&
  TRIGGER_FIRED_BEFORE(trigdata->tg_event))
  checknull = true;

 if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
  when = "before";
 else
  when = "after ";

 tupdesc = trigdata->tg_relation->rd_att;

 /* Connect to SPI manager */
 if ((ret = SPI_connect()) < 0)
```

```
 elog(NOTICE, "trigf (fired %s): SPI_connect returned %d", when,
ret);

/* Get number of tuples in relation */
ret = SPI_exec("SELECT count(*) FROM ttest", 0);

if (ret < 0)
 elog(NOTICE, "trigf (fired %s): SPI_exec returned %d", when, ret);

/* count(*) returns int8 as of PG 7.2, so be careful to convert */
i = (int) DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
          SPI_tuptable->tupdesc,
          1,
          &isnull));

elog (NOTICE, "trigf (fired %s): there are %d tuples in ttest", when,
i);

SPI_finish();

if (checknull)
{
 (void) SPI_getbinval(rettuple, tupdesc, 1, &isnull);
 if (isnull)
  rettuple = NULL;
}

return PointerGetDatum(rettuple);
}
```

Now, compile and create the trigger function:

```
CREATE FUNCTION trigf () RETURNS OPAQUE AS
'...path_to_so' LANGUAGE 'C';

CREATE TABLE ttest (x int4);


vac=> CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON
 ttest
FOR EACH ROW EXECUTE PROCEDURE trigf();
CREATE
vac=> CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
FOR EACH ROW EXECUTE PROCEDURE trigf();
CREATE
vac=> INSERT INTO ttest VALUES (NULL);
NOTICE:trigf (fired before): there are 0 tuples in ttest
INSERT 0 0

-- Insertion skipped and AFTER trigger is not fired

vac=> SELECT * FROM ttest;
x
-
```

```
(0 rows)

vac=> INSERT INTO ttest VALUES (1);
NOTICE:trigf (fired before): there are 0 tuples in ttest
NOTICE:trigf (fired after ): there are 1 tuples in ttest
                                       ^^^^^^^^
                             remember what we said about visibility.
INSERT 167793 1
vac=> SELECT * FROM ttest;
x
-
1
(1 row)

vac=> INSERT INTO ttest SELECT x * 2 FROM ttest;
NOTICE:trigf (fired before): there are 1 tuples in ttest
NOTICE:trigf (fired after ): there are 2 tuples in ttest
                                       ^^^^^^^^
                             remember what we said about visibility.
INSERT 167794 1
vac=> SELECT * FROM ttest;
x
-
1
2
(2 rows)

vac=> UPDATE ttest SET x = null WHERE x = 2;
NOTICE:trigf (fired before): there are 2 tuples in ttest
UPDATE 0
vac=> UPDATE ttest SET x = 4 WHERE x = 2;
NOTICE:trigf (fired before): there are 2 tuples in ttest
NOTICE:trigf (fired after ): there are 2 tuples in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
x
-
1
4
(2 rows)

vac=> DELETE FROM ttest;
NOTICE:trigf (fired before): there are 2 tuples in ttest
NOTICE:trigf (fired after ): there are 1 tuples in ttest
NOTICE:trigf (fired before): there are 1 tuples in ttest
NOTICE:trigf (fired after ): there are 0 tuples in ttest
                                       ^^^^^^^^
                             remember what we said about visibility.
DELETE 2
vac=> SELECT * FROM ttest;
x
-
(0 rows)
```

Chapter 21. Server Programming Interface

Vadim Mikheev

The *Server Programming Interface* (SPI) gives users the ability to run SQL queries inside user-defined C functions.

Note

The available Procedural Languages (PL) give an alternate means to build functions that can execute queries.

In fact, SPI is just a set of native interface functions to simplify access to the Parser, Planner, Optimizer and Executor. SPI also does some memory management.

To avoid misunderstanding we'll use *function* to mean SPI interface functions and *procedure* for user-defined C-functions using SPI.

Procedures which use SPI are called by the Executor. The SPI calls recursively invoke the Executor in turn to run queries. When the Executor is invoked recursively, it may itself call procedures which may make SPI calls.

Note that if during execution of a query from a procedure the transaction is aborted, then control will not be returned to your procedure. Rather, all work will be rolled back and the server will wait for the next command from the client. This will probably be changed in future versions.

A related restriction is the inability to execute BEGIN, END and ABORT (transaction control statements). This will also be changed in the future.

If successful, SPI functions return a non-negative result (either via a returned integer value or in SPI_result global variable, as described below). On error, a negative or NULL result will be returned.

21.1. Interface Functions

Name

SPI_connect — Connects your procedure to the SPI manager.

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
int SPI_connect(void)
```

Inputs

None

Outputs

int

Return status

SPI_OK_CONNECT

if connected

SPI_ERROR_CONNECT

if not connected

Description

`SPI_connect` opens a connection from a procedure invocation to the SPI manager. You must call this function if you will need to execute queries. Some utility SPI functions may be called from un-connected procedures.

If your procedure is already connected, `SPI_connect` will return an SPI_ERROR_CONNECT error. Note that this may happen if a procedure which has called `SPI_connect` directly calls another procedure which itself calls `SPI_connect`. While recursive calls to the SPI manager are permitted when an SPI query invokes another function which uses SPI, directly nested calls to `SPI_connect` and `SPI_finish` are forbidden.

Usage

Algorithm

`SPI_connect` performs the following: Initializes the SPI internal structures for query execution and memory management.

Name

SPI_finish — Disconnects your procedure from the SPI manager.

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_finish(void)
```

Inputs

None

Outputs

int

SPI_OK_FINISH if properly disconnected
SPI_ERROR_UNCONNECTED if called from an un-connected procedure

Description

`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 may get the error return SPI_ERROR_UNCONNECTED if `SPI_finish` is called without having a current valid connection. There is no fundamental problem with this; it means that nothing was done by the SPI manager.

Usage

`SPI_finish` *must* be called as a final step by a connected procedure, or you may get unpredictable results! However, you do not need to worry about making this happen if the transaction is aborted via elog(ERROR). In that case SPI will clean itself up.

Algorithm

`SPI_finish` performs the following: Disconnects your procedure from the SPI manager and frees all memory allocations made by your procedure via `palloc` since the `SPI_connect`. These allocations can't be used any more! See Memory management.

Name

SPI_exec — Creates an execution plan (parser+planner+optimizer) and executes a query.

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_exec(query, tcount)
```

Inputs

char *query

String containing query plan

int tcount

Maximum number of tuples to return

Outputs

int

SPI_ERROR_UNCONNECTED if called from an un-connected procedure
SPI_ERROR_ARGUMENT if query is NULL or tcount < 0.
SPI_ERROR_UNCONNECTED if procedure is unconnected.
SPI_ERROR_COPY if COPY TO/FROM stdin.
SPI_ERROR_CURSOR if DECLARE/CLOSE CURSOR, FETCH.
SPI_ERROR_TRANSACTION if BEGIN/ABORT/END.
SPI_ERROR_OPUNKNOWN if type of query is unknown (this shouldn't occur).

If execution of your query was successful then one of the following (non-negative) values will be returned:

SPI_OK_UTILITY if some utility (e.g. CREATE TABLE ...) was executed
SPI_OK_SELECT if SELECT (but not SELECT ... INTO!) was executed
SPI_OK_SELINTO if SELECT ... INTO was executed
SPI_OK_INSERT if INSERT (or INSERT ... SELECT) was executed
SPI_OK_DELETE if DELETE was executed
SPI_OK_UPDATE if UPDATE was executed

Description

SPI_exec creates an execution plan (parser+planner+optimizer) and executes the query for tcount tuples.

Usage

This should only be called from a connected procedure. If tcount is zero then it executes the query for all tuples returned by the query scan. Using tcount > 0 you may restrict the number of tuples for which the query will be executed (much like a LIMIT clause). For example,

```
SPI_exec ("INSERT INTO tab SELECT * FROM tab", 5);
```

will allow at most 5 tuples to be inserted into table. If execution of your query was successful then a non-negative value will be returned.

Note

You may pass multiple queries in one string or query string may be re-written by RULEs. `SPI_exec` returns the result for the last query executed.

The actual number of tuples for which the (last) query was executed is returned in the global variable SPI_processed (if not SPI_OK_UTILITY). If SPI_OK_SELECT is returned and SPI_processed > 0 then you may use global pointer SPITupleTable *SPI_tuptable to access the result tuples.

`SPI_exec` may return one of the following (negative) values:

SPI_ERROR_ARGUMENT if query is NULL or *`tcount`* < 0.
SPI_ERROR_UNCONNECTED if procedure is unconnected.
SPI_ERROR_COPY if COPY TO/FROM stdin.
SPI_ERROR_CURSOR if DECLARE/CLOSE CURSOR, FETCH.
SPI_ERROR_TRANSACTION if BEGIN/ABORT/END.
SPI_ERROR_OPUNKNOWN if type of query is unknown (this shouldn't occur).

Structures

If SPI_OK_SELECT is returned and SPI_processed > 0 then you may use the global pointer SPITupleTable *SPI_tuptable to access the selected tuples.

Structure SPITupleTable is defined in spi.h:

```
typedef struct
{
    MemoryContext tuptabcxt;    /* memory context of result table
 */
    uint32      alloced;        /* # of alloced vals */
    uint32      free;           /* # of free vals */
    TupleDesc   tupdesc;        /* tuple descriptor */
    HeapTuple  *vals;           /* tuples */
} SPITupleTable;
```

vals is an array of pointers to tuples (the number of useful entries is given by SPI_processed). TupleDesc tupdesc is a tuple descriptor which you may pass to SPI functions dealing with tuples. tuptabcxt, alloced, and free are internal fields not intended for use by SPI callers.

Note

Functions `SPI_exec`, `SPI_execp` and `SPI_prepare` 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 of one `SPI_exec` or `SPI_execp` across later calls.

`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`.

Name

SPI_prepare — Prepares a plan for a query, without executing it yet

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_prepare(query, nargs, argtypes)
```

Inputs

query

Query string

nargs

Number of input parameters ($1 ... $nargs - as in SQL-functions)

argtypes

Pointer to array of type OIDs for input parameter types

Outputs

void *

Pointer to an execution plan (parser+planner+optimizer)

Description

`SPI_prepare` creates and returns an execution plan (parser+planner+optimizer) but doesn't execute the query. Should only be called from a connected procedure.

Usage

When the same or similar query is to be executed repeatedly, it may be advantageous to perform query planning only once. `SPI_prepare` converts a query string into an execution plan that can be passed repeatedly to `SPI_execp`.

A prepared query can be generalized by writing parameters ($1, $2, etc) in place of what would be constants in a normal query. The values of the parameters are then specified when `SPI_execp` is called. This allows the prepared query to be used over a wider range of situations than would be possible without parameters.

Note

However, there is a disadvantage: since the planner does not know the values that will be supplied for the parameters, it may make worse query planning choices than it would make for a simple query with all constants visible.

If the query uses parameters, their number and datatypes must be specified in the call to `SPI_prepare`.

The plan returned by `SPI_prepare` may be used only in current invocation of the procedure since `SPI_finish` frees memory allocated for a plan. But see `SPI_saveplan` to save a plan for longer.

If successful, a non-null pointer will be returned. Otherwise, you'll get a NULL plan. In both cases SPI_result will be set like the value returned by SPI_exec, except that it is set to SPI_ERROR_ARGUMENT if query is NULL or nargs < 0 or nargs > 0 && argtypes is NULL.

Name

SPI_execp — Executes a plan from `SPI_prepare`

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_execp(plan,
values,
nulls,
tcount)
```

Inputs

void **plan*

Execution plan

Datum **values*

Actual parameter values

char **nulls*

Array describing which parameters are NULLs

`n` indicates NULL (values[] entry ignored)
space indicates not NULL (values[] entry is valid)

int *tcount*

Number of tuples for which plan is to be executed

Outputs

int

Returns the same value as `SPI_exec` as well as

SPI_ERROR_ARGUMENT if *plan* is NULL or *tcount* < 0
SPI_ERROR_PARAM if *values* is NULL and *plan* was prepared with some parameters.

SPI_tuptable

initialized as in `SPI_exec` if successful

SPI_processed

initialized as in `SPI_exec` if successful

Description

`SPI_execp` executes a plan prepared by `SPI_prepare`. *tcount* has the same interpretation as in `SPI_exec`.

Usage

If *nulls* is NULL then `SPI_execp` assumes that all parameters (if any) are NOT NULL.

Note

If one of the objects (a relation, function, etc.) referenced by the prepared plan is dropped during your session (by your backend or another process) then the results of `SPI_execp` for this plan will be unpredictable.

Name

SPI_cursor_open — Sets up a cursor using a plan created with `SPI_prepare`

Synopsis

<refsynopsisdivinfo>2001-11-14</refsynopsisdivinfo>

```
SPI_cursor_open(name,
plan,
values,
nulls)
```

Inputs

char **name*

Name for portal, or NULL to let the system select a name

void **plan*

Execution plan

Datum **values*

Actual parameter values

char **nulls*

Array describing which parameters are NULLs

`n` indicates NULL (values[] entry ignored)
space indicates not NULL (values[] entry is valid)

Outputs

Portal

Pointer to Portal containing cursor, or NULL on error

Description

`SPI_cursor_open` sets up a cursor (internally, a Portal) that will execute a plan prepared by `SPI_prepare`.

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 rowset result.

Usage

If *nulls* is NULL then `SPI_cursor_open` assumes that all parameters (if any) are NOT NULL.

Name

SPI_cursor_find — Finds an existing cursor (Portal) by name

Synopsis

<refsynopsisdivinfo>2001-11-14</refsynopsisdivinfo>

```
SPI_cursor_find(name)
```

Inputs

char **name*

Name of portal

Outputs

Portal

Pointer to Portal with given name, or NULL if not found

Description

`SPI_cursor_find` finds a pre-existing Portal by name. This is primarily useful to resolve a cursor name returned as text by some other function.

Name

SPI_cursor_fetch — Fetches some rows from a cursor

Synopsis

<refsynopsisdivinfo>2001-11-14</refsynopsisdivinfo>

```
SPI_cursor_fetch(portal,
forward,
count)
```

Inputs

Portal *portal*

Portal containing cursor

bool *forward*

True for fetch forward, false for fetch backward

int *count*

Maximum number of rows to fetch

Outputs

SPI_tuptable

initialized as in `SPI_exec` if successful

SPI_processed

initialized as in `SPI_exec` if successful

Description

`SPI_cursor_fetch` fetches some (more) rows from a cursor. This is equivalent to the SQL command `FETCH`.

Name

SPI_cursor_move — Moves a cursor

Synopsis

<refsynopsisdivinfo>2001-11-14</refsynopsisdivinfo>

```
SPI_cursor_move(portal,
forward,
count)
```

Inputs

Portal *portal*

Portal containing cursor

bool *forward*

True for move forward, false for move backward

int *count*

Maximum number of rows to move

Outputs

None

Description

`SPI_cursor_move` skips over some number of rows in a cursor. This is equivalent to the SQL command `MOVE`.

Name

SPI_cursor_close — Closes a cursor

Synopsis

<refsynopsisdivinfo>2001-11-14</refsynopsisdivinfo>

```
SPI_cursor_close(portal)
```

Inputs

Portal *portal*

Portal containing cursor

Outputs

None

Description

`SPI_cursor_close` closes a previously created cursor and releases its Portal storage.

Usage

All open cursors are closed implicitly at transaction end. `SPI_cursor_close` need only be invoked if it is desirable to release resources sooner.

Name

SPI_saveplan — Saves a passed plan

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_saveplan(plan)
```

Inputs

void **query*

Passed plan

Outputs

void *

Execution plan location. NULL if unsuccessful.

SPI_result

SPI_ERROR_ARGUMENT if plan is NULL
SPI_ERROR_UNCONNECTED if procedure is un-connected

Description

`SPI_saveplan` stores a plan prepared by `SPI_prepare` in safe memory protected from freeing by `SPI_finish` or the transaction manager.

In the current version of PostgreSQL there is no ability to store prepared plans in the system catalog and fetch them from there for execution. This will be implemented in future versions. As an alternative, there is the ability to reuse prepared plans in the subsequent invocations of your procedure in the current session. Use `SPI_execp` to execute this saved plan.

Usage

`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. 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_execp (see below).

Note

If one of the objects (a relation, function, etc.) referenced by the prepared plan is dropped during your session (by your backend or another process) then the results of `SPI_execp` for this plan will be unpredictable.

21.2. Interface Support Functions

The functions described here provide convenient interfaces for extracting information from tuple sets returned by `SPI_exec` and other SPI interface functions.

All functions described in this section may be used by both connected and unconnected procedures.

Name

SPI_fnumber — Finds the attribute number for specified attribute name

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_fnumber(tupdesc, fname)
```

Inputs

TupleDesc *tupdesc*

Input tuple description

char * *fname*

Field name

Outputs

int

Attribute number

Valid one-based index number of attribute
SPI_ERROR_NOATTRIBUTE if the named attribute is not found

Description

`SPI_fnumber` returns the attribute number for the attribute with name in fname.

Usage

Attribute numbers are 1 based.

If the given fname refers to a system attribute (eg, `oid`) then the appropriate negative attribute number will be returned. The caller should be careful to test for exact equality to SPI_ERROR_NOATTRIBUTE to detect error; testing for result <= 0 is not correct unless system attributes should be rejected.

Name

SPI_fname — Finds the attribute name for the specified attribute number

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_fname(tupdesc, fnumber)
```

Inputs

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

char *

Attribute name

NULL if fnumber is out of range
SPI_result set to SPI_ERROR_NOATTRIBUTE on error

Description

`SPI_fname` returns the attribute name for the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

Returns a newly-allocated copy of the attribute name. (Use pfree() to release the copy when done with it.)

Name

SPI_getvalue — Returns the string value of the specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_getvalue(tuple, tupdesc, fnumber)
```

Inputs

HeapTuple *tuple*

Input tuple to be examined

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

char *

Attribute value or NULL if

attribute is NULL
fnumber is out of range (SPI_result set to SPI_ERROR_NOATTRIBUTE)
no output function available (SPI_result set to SPI_ERROR_NOOUTFUNC)

Description

`SPI_getvalue` returns an external (string) representation of the value of the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

The result is returned as a palloc'd string. (Use pfree() to release the string when done with it.)

Name

SPI_getbinval — Returns the binary value of the specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_getbinval(tuple, tupdesc, fnumber, isnull)
```

Inputs

HeapTuple *tuple*

Input tuple to be examined

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

Datum

Attribute binary value

bool * *isnull*

flag for null value in attribute

SPI_result

SPI_ERROR_NOATTRIBUTE

Description

`SPI_getbinval` returns the specified attribute's value in internal form (as a Datum).

Usage

Attribute numbers are 1 based.

Algorithm

Does not allocate new space for the datum. In the case of a pass-by- reference datatype, the Datum will be a pointer into the given tuple.

Name

SPI_gettype — Returns the type name of the specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_gettype(tupdesc, fnumber)
```

Inputs

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

char *

The type name for the specified attribute number

SPI_result

SPI_ERROR_NOATTRIBUTE

Description

`SPI_gettype` returns a copy of the type name for the specified attribute, or NULL on error.

Usage

Attribute numbers are 1 based.

Algorithm

Returns a newly-allocated copy of the type name. (Use pfree() to release the copy when done with it.)

Name

SPI_gettypeid — Returns the type OID of the specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_gettypeid(tupdesc, fnumber)
```

Inputs

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

OID

The type OID for the specified attribute number

SPI_result

SPI_ERROR_NOATTRIBUTE

Description

`SPI_gettypeid` returns the type OID for the specified attribute.

Usage

Attribute numbers are 1 based.

Name

SPI_getrelname — Returns the name of the specified relation

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_getrelname(rel)
```

Inputs

Relation *rel*

Input relation

Outputs

char *

The name of the specified relation

Description

`SPI_getrelname` returns the name of the specified relation.

Algorithm

Returns a newly-allocated copy of the rel name. (Use pfree() to release the copy when done with it.)

21.3. 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 --- 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 allocated memory object (such as a value of a pass-by-reference datatype), you can't allocate the return object using `palloc`, at least not while you are connected to SPI. If you try, the object will be deallocated during `SPI_finish`, and your procedure will not work reliably!

To solve this problem, use `SPI_palloc` to allocate your return object. `SPI_palloc` allocates space from "upper Executor" memory --- that is, the memory context that was current when `SPI_connect` was called, which is precisely the right context for return values of your procedure.

If called while not connected to SPI, `SPI_palloc` acts the same as plain `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.

After SPI_connect is called, the current context is the procedure's private context made by SPI_connect. All allocations made via palloc/repalloc or by SPI utility functions (except for SPI_copytuple, SPI_copytupledesc, SPI_copytupleintoslot, 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 can't 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 backend functions (palloc etc).

Name

SPI_copytuple — Makes copy of tuple in upper Executor context

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_copytuple(tuple)
```

Inputs

HeapTuple *tuple*

Input tuple to be copied

Outputs

HeapTuple

Copied tuple

non-NULL if *tuple* is not NULL and the copy was successful
NULL only if *tuple* is NULL

Description

`SPI_copytuple` makes a copy of tuple in upper Executor context.

Usage

TBD

Name

SPI_copytupledesc — Makes copy of tuple descriptor in upper Executor context

Synopsis

<refsynopsisdivinfo>2001-08-02</refsynopsisdivinfo>

```
SPI_copytupledesc(tupdesc)
```

Inputs

TupleDesc *tupdesc*

Input tuple descriptor to be copied

Outputs

TupleDesc

Copied tuple descriptor

non-NULL if *tupdesc* is not NULL and the copy was successful
NULL only if *tupdesc* is NULL

Description

`SPI_copytupledesc` makes a copy of tupdesc in upper Executor context.

Usage

TBD

Name

SPI_copytupleintoslot — Makes copy of tuple and descriptor in upper Executor context

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_copytupleintoslot(tuple, tupdesc)
```

Inputs

HeapTuple *tuple*

Input tuple to be copied

TupleDesc *tupdesc*

Input tuple descriptor to be copied

Outputs

TupleTableSlot *

Tuple slot containing copied tuple and descriptor

non-NULL if *tuple* and *tupdesc* are not NULL and the copy was successful
NULL only if *tuple* or *tupdesc* is NULL

Description

`SPI_copytupleintoslot` makes a copy of tuple in upper Executor context, returning it in the form of a filled-in TupleTableSlot.

Usage

TBD

Name

SPI_modifytuple — Creates a tuple by replacing selected fields of a given tuple

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_modifytuple(rel, tuple, nattrs, attnum, Values, Nulls)
```

Inputs

Relation *rel*

Used only as source of tuple descriptor for tuple. (Passing a relation rather than a tuple descriptor is a misfeature.)

HeapTuple *tuple*

Input tuple to be modified

int *nattrs*

Number of attribute numbers in attnum array

int * *attnum*

Array of numbers of the attributes that are to be changed

Datum * *Values*

New values for the attributes specified

char * *Nulls*

Which new values are NULL, if any

Outputs

HeapTuple

New tuple with modifications

non-NULL if *tuple* is not NULL and the modify was successful
NULL only if *tuple* is NULL

SPI_result

SPI_ERROR_ARGUMENT if rel is NULL or tuple is NULL or natts <= 0 or attnum is NULL or Values is NULL.
SPI_ERROR_NOATTRIBUTE if there is an invalid attribute number in attnum (attnum <= 0 or > number of attributes in tuple)

Description

`SPI_modifytuple` creates a new tuple by substituting new values for selected attributes, copying the original tuple's attributes at other positions. The input tuple is not modified.

Usage

If successful, a pointer to the new tuple is returned. The new tuple is allocated in upper Executor context.

Name

SPI_palloc — Allocates memory in upper Executor context

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_palloc(size)
```

Inputs

Size *size*

Octet size of storage to allocate

Outputs

void *

New storage space of specified size

Description

`SPI_palloc` allocates memory in upper Executor context.

Usage

TBD

Name

SPI_repalloc — Re-allocates memory in upper Executor context

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_repalloc(pointer, size)
```

Inputs

void * *pointer*

Pointer to existing storage

Size *size*

Octet size of storage to allocate

Outputs

void *

New storage space of specified size with contents copied from existing area

Description

`SPI_repalloc` re-allocates memory in upper Executor context.

Usage

This function is no longer different from plain `repalloc`. It's kept just for backward compatibility of existing code.

Name

SPI_pfree — Frees memory in upper Executor context

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_pfree(pointer)
```

Inputs

void * *pointer*

Pointer to existing storage

Outputs

None

Description

`SPI_pfree` frees memory in upper Executor context.

Usage

This function is no longer different from plain `pfree`. It's kept just for backward compatibility of existing code.

Name

SPI_freetuple — Frees a tuple allocated in upper Executor context

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_freetuple(pointer)
```

Inputs

HeapTuple *pointer*

Pointer to allocated tuple

Outputs

None

Description

`SPI_freetuple` frees a tuple previously allocated in upper Executor context.

Usage

This function is no longer different from plain `heap_freetuple`. It's kept just for backward compatibility of existing code.

Name

SPI_freetuptable — Frees a tuple set created by `SPI_exec` or similar function

Synopsis

<refsynopsisdivinfo>2001-11-14</refsynopsisdivinfo>

```
SPI_freetuptable(tuptable)
```

Inputs

SPITupleTable * *`tuptable`*

Pointer to tuple table

Outputs

None

Description

`SPI_freetuptable` frees a tuple set created by a prior SPI query function, such as `SPI_exec`.

Usage

This function is useful if a SPI procedure needs to execute multiple queries and does not want to keep the results of earlier queries around until it ends. Note that any unfreed tuple sets will be freed anyway at `SPI_finish`.

Name

SPI_freeplan — Releases a previously saved plan

Synopsis

<refsynopsisdivinfo>2001-11-14</refsynopsisdivinfo>

```
SPI_freeplan(plan)
```

Inputs

void **plan*

Passed plan

Outputs

int

SPI_ERROR_ARGUMENT if plan is NULL

Description

`SPI_freeplan` releases a query plan previously returned by `SPI_prepare` or saved by `SPI_saveplan`.

21.4. Visibility of Data Changes

PostgreSQL data changes visibility rule: during a query execution, data changes made by the query itself (via SQL-function, SPI-function, triggers) are invisible to the query scan. For example, in query

```
INSERT INTO a SELECT * FROM a
```

tuples inserted are invisible for SELECT's scan. In effect, this duplicates the database table within itself (subject to unique index rules, of course) without recursing.

Changes made by query Q are visible to queries that are started after query Q, no matter whether they are started inside Q (during the execution of Q) or after Q is done.

21.5. Examples

This example of SPI usage demonstrates the visibility rule. There are more complex examples in src/test/regress/regress.c and in contrib/spi.

This is a very simple example of SPI usage. The procedure execq accepts an SQL-query in its first argument and tcount in its second, executes the query using SPI_exec and returns the number of tuples for which the query executed:

```
#include "executor/spi.h"   /* this is what you need to work with SPI
 */

int execq(text *sql, int cnt);

int
```

```
execq(text *sql, int cnt)
{
    char *query;
    int ret;
    int proc;

    /* Convert given TEXT object to a C string */
    query = DatumGetCString(DirectFunctionCall1(textout,

 PointerGetDatum(sql)));

    SPI_connect();

    ret = SPI_exec(query, cnt);

    proc = SPI_processed;
    /*
     * If this is SELECT and some tuple(s) fetched -
     * returns tuples to the caller via elog (NOTICE).
     */
    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++)
                sprintf(buf + strlen (buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog (NOTICE, "EXECQ: %s", buf);
        }
    }

    SPI_finish();

    pfree(query);

    return (proc);
}
```

Now, compile and create the function:

```
CREATE FUNCTION execq (text, integer) RETURNS integer
    AS '...path_to_so'
    LANGUAGE C;

vac=> SELECT execq('CREATE TABLE a (x INTEGER)', 0);
execq
```

```
-----
    0
(1 row)

vac=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)',0));
INSERT 167631 1
vac=> SELECT execq('SELECT * FROM a',0);
NOTICE:EXECQ:  0 <<< inserted by execq

NOTICE:EXECQ:  1 <<< value returned by execq and inserted by upper
 INSERT

execq
-----
    2
(1 row)

vac=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a',1);
execq
-----
    1
(1 row)

vac=> SELECT execq('SELECT * FROM a', 10);
NOTICE:EXECQ:  0

NOTICE:EXECQ:  1

NOTICE:EXECQ:  2 <<< 0 + 2, only one tuple inserted - as specified

execq
-----
    3            <<< 10 is max value only, 3 is real # of tuples
(1 row)

vac=> DELETE FROM a;
DELETE 3
vac=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 167712 1
vac=> SELECT * FROM a;
x
-
1                <<< no tuples in a (0) + 1
(1 row)

vac=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
NOTICE:EXECQ:  0
INSERT 167713 1
vac=> SELECT * FROM a;
x
-
1
2                <<< there was single tuple in a + 1
(2 rows)
```

```
--   This demonstrates data changes visibility rule:

vac=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
NOTICE:EXECQ:  1
NOTICE:EXECQ:  2
NOTICE:EXECQ:  1
NOTICE:EXECQ:  2
NOTICE:EXECQ:  2
INSERT 0 2
vac=> SELECT * FROM a;
x
-
1
2
2               <<< 2 tuples * 1 (x in first tuple)
6               <<< 3 tuples (2 + 1 just inserted) * 2 (x in second
 tuple)
(4 rows)            ^^^^^^^^
                    tuples visible to execq() in different
 invocations
```

Part III. Procedural Languages

This part documents the procedural languages available in the PostgreSQL distribution as well as general issues concerning procedural languages.

Table of Contents

22. Procedural Languages 292
    22.1. Introduction 292
    22.2. Installing Procedural Languages 292
23. PL/pgSQL - SQL Procedural Language 294
    23.1. Overview 294
        23.1.1. Advantages of Using PL/pgSQL 295
        23.1.2. Developing in PL/pgSQL 295
    23.2. Structure of PL/pgSQL 296
        23.2.1. Lexical Details 297
    23.3. Declarations 297
        23.3.1. Aliases for Function Parameters 298
        23.3.2. Rowtypes 298
        23.3.3. Records 298
        23.3.4. Attributes 299
        23.3.5. RENAME 299
    23.4. Expressions 300
    23.5. Basic Statements 301
        23.5.1. Assignment 301
        23.5.2. SELECT INTO 301
        23.5.3. Executing an expression or query with no result 302
        23.5.4. Executing dynamic queries 303
        23.5.5. Obtaining result status 304
    23.6. Control Structures 304
        23.6.1. Returning from a function 304
        23.6.2. Conditionals 305
        23.6.3. Simple Loops 306
        23.6.4. Looping Through Query Results 308
    23.7. Cursors 309
        23.7.1. Declaring Cursor Variables 309
        23.7.2. Opening Cursors 310
        23.7.3. Using Cursors 310
    23.8. Errors and Messages 311
        23.8.1. Exceptions 311
    23.9. Trigger Procedures 312
    23.10. Examples 314
    23.11. Porting from Oracle PL/SQL 315
        23.11.1. Main Differences 315
        23.11.2. Porting Functions 316
        23.11.3. Procedures 320
        23.11.4. Packages 322
        23.11.5. Other Things to Watch For 323
        23.11.6. Appendix 323
24. PL/Tcl - Tcl Procedural Language 327
    24.1. Overview 327
    24.2. Description 327
        24.2.1. PL/Tcl Functions and Arguments 327
        24.2.2. Data Values in PL/Tcl 328
        24.2.3. Global Data in PL/Tcl 329
        24.2.4. Database Access from PL/Tcl 329
        24.2.5. Trigger Procedures in PL/Tcl 331
        24.2.6. Modules and the `unknown` command 333

24.2.7. Tcl Procedure Names 333
25. PL/Perl - Perl Procedural Language 334
25.1. Overview 334
25.2. Building and Installing PL/Perl 334
25.3. Description 335
25.3.1. PL/Perl Functions and Arguments 335
25.3.2. Data Values in PL/Perl 336
25.3.3. Database Access from PL/Perl 336
25.3.4. Missing Features 337
26. PL/Python - Python Procedural Language 338
26.1. Introduction 338
26.2. Installation 338
26.3. Using PL/Python 338

Chapter 22. Procedural Languages

22.1. Introduction

PostgreSQL allows users to add new programming languages to be available for writing functions and procedures. These are called *procedural languages* (PL). In the case of a function or trigger procedure written in a procedural language, the database server has no built-in knowledge about how to interpret the function's source text. Instead, the task is passed to a special handler that knows the details of the language. The handler could either do all the work of parsing, syntax analysis, execution, etc. itself, or it could serve as "glue" between PostgreSQL and an existing implementation of a programming language. The handler itself is a special programming language function compiled into a shared object and loaded on demand.

Writing a handler for a new procedural language is described in Section 12.7, "Procedural Language Handlers". Several procedural languages are available in the standard PostgreSQL distribution, which can serve as examples.

22.2. Installing Procedural Languages

A procedural language must be "installed" into each database where it is to be used. But procedural languages installed in the template1 database are automatically available in all subsequently created databases. So the database administrator can decide which languages are available in which databases, and can make some languages available by default if he chooses.

For the languages supplied with the standard distribution, the shell script `createlang` may be used instead of carrying out the details by hand. For example, to install PL/pgSQL into the template1 database, use

```
createlang plpgsql template1
```

The manual procedure described below is only recommended for installing custom languages that `createlang` does not know about.

Manual Procedural Language Installation

A procedural language is installed in the database in three steps, which must be carried out by a database superuser.

1. The shared object for the language handler must be compiled and installed into an appropriate library directory. This works in the same way as building and installing modules with regular user-defined C functions does; see Section 12.5.7, "Compiling and Linking Dynamically-Loaded Functions".

2. The handler must be declared with the command

   ```
   CREATE FUNCTION handler_function_name ()
       RETURNS OPAQUE AS
       'path-to-shared-object' LANGUAGE C;
   ```

   The special return type of `OPAQUE` tells the database that this function does not return one of the defined SQL data types and is not directly usable in SQL statements.

3. The PL must be declared with the command

   ```
   CREATE [TRUSTED] [PROCEDURAL] LANGUAGE language-name
   ```

```
HANDLER handler_function_name;
```

The optional key word `TRUSTED` tells whether ordinary database users that have no superuser privileges should be allowed to use this language to create functions and trigger procedures. Since PL functions are executed inside the database server, the `TRUSTED` flag should only be given for languages that do not allow access to database server internals or the file system. The languages PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python are known to be trusted; the languages PL/TclU and PL/PerlU are designed to provide unlimited functionality should *not* be marked trusted.

In a default PostgreSQL installation, the handler for the PL/pgSQL language is built and installed into the "library" directory. If Tcl/Tk support is configured in, the handlers for PL/Tcl and PL/TclU are also built and installed in the same location. Likewise, the PL/Perl and PL/PerlU handlers are built and installed if Perl support is configured, and PL/Python is installed if Python support is configured. The `createlang` script automates Step 2 and Step 3 described above.

Example 22.1. Manual Installation of PL/pgSQL

The following command tells the database server where to find the shared object for the PL/pgSQL language's call handler function.

```
CREATE FUNCTION plpgsql_call_handler () RETURNS OPAQUE AS
    '$libdir/plpgsql' LANGUAGE C;
```

The command

```
CREATE TRUSTED PROCEDURAL LANGUAGE plpgsql
    HANDLER plpgsql_call_handler;
```

then defines that the previously declared call handler function should be invoked for functions and trigger procedures where the language attribute is `plpgsql`.

Chapter 23. PL/pgSQL - SQL Procedural Language

PL/pgSQL is a loadable procedural language for the PostgreSQL database system.

This package was originally written by Jan Wieck. This documentation was in part written by Roberto Mello (<rmello@fslc.usu.edu>).

23.1. Overview

The design goals of PL/pgSQL were to create a loadable procedural language that

- can be used to create functions and trigger procedures,
- adds control structures to the SQL language,
- can perform complex computations,
- inherits all user defined types, functions and operators,
- can be defined to be trusted by the server,
- is easy to use.

The PL/pgSQL call handler parses the function's source text and produces an internal binary instruction tree the first time the function is called (within any one backend process). The instruction tree fully translates the PL/pgSQL statement structure, but individual SQL expressions and SQL queries used in the function are not translated immediately.

As each expression and SQL query is first used in the function, the PL/pgSQL interpreter creates a prepared execution plan (using the SPI manager's `SPI_prepare` and `SPI_saveplan` functions). Subsequent visits to that expression or query re-use the prepared plan. Thus, a function with conditional code that contains many statements for which execution plans might be required, will only prepare and save those plans that are really used during the lifetime of the database connection. This can provide a considerable savings of parsing activity. A disadvantage is that errors in a specific expression or query may not be detected until that part of the function is reached in execution.

Once PL/pgSQL has made a query plan for a particular query in a function, it will re-use that plan for the life of the database connection. This is usually a win for performance, but it can cause some problems if you dynamically alter your database schema. For example:

```
CREATE FUNCTION populate() RETURNS INTEGER AS '
DECLARE
    -- Declarations
BEGIN
    PERFORM my_function();
END;
' LANGUAGE 'plpgsql';
```

If you execute the above function, it will reference the OID for `my_function()` in the query plan produced for the PERFORM statement. Later, if you drop and re-create `my_function()`, then `populate()` will not be able to find `my_function()` anymore. You would then have to re-create `populate()`, or at least start a new database session so that it will be compiled afresh.

Because PL/pgSQL saves execution plans in this way, queries that appear directly in a PL/pgSQL function must refer to the same tables and fields on every execution; that is, you cannot use a parameter as the name of a table or field in a query. To get around this restriction, you can construct dynamic queries using the PL/pgSQL EXECUTE statement --- at the price of constructing a new query plan on every execution.

Except for input/output conversion and calculation functions for user defined types, anything that can be defined in C language functions can also be done with PL/pgSQL. It is possible to create complex conditional computation functions and later use them to define operators or use them in functional indexes.

23.1.1. Advantages of Using PL/pgSQL

- Better performance (see Section 23.1.1.1, “Better Performance”)
- SQL support (see Section 23.1.1.2, “SQL Support”)
- Portability (see Section 23.1.1.3, “Portability”)

23.1.1.1. Better Performance

SQL is the language PostgreSQL (and most other Relational Databases) use as query language. It's portable and easy to learn. But every SQL statement must be executed individually by the database server.

That means that your client application must send each query to the database server, wait for it to process it, receive the results, do some computation, then send other queries to the server. All this incurs inter-process communication and may also incur network overhead if your client is on a different machine than the database server.

With PL/pgSQL you can group a block of computation and a series of queries *inside* the database server, thus having the power of a procedural language and the ease of use of SQL, but saving lots of time because you don't have the whole client/server communication overhead. This can make for a considerable performance increase.

23.1.1.2. SQL Support

PL/pgSQL adds the power of a procedural language to the flexibility and ease of SQL. With PL/pgSQL you can use all the data types, columns, operators and functions of SQL.

23.1.1.3. Portability

Because PL/pgSQL functions run inside PostgreSQL, these functions will run on any platform where PostgreSQL runs. Thus you can reuse code and have less development costs.

23.1.2. Developing in PL/pgSQL

Developing in PL/pgSQL is pretty straight forward, especially if you have developed in other database procedural languages, such as Oracle's PL/SQL. Two good ways of developing in PL/pgSQL are:

- Using a text editor and reloading the file with `psql`
- Using PostgreSQL's GUI Tool: PgAccess

One good way to develop in PL/pgSQL is to simply use the text editor of your choice to create your functions, and in another console, use `psql` (PostgreSQL's interactive monitor) to load those functions. If you are doing it this way, it is a good idea to write the function using `CREATE OR REPLACE FUNCTION`. That way you can reload the file to update the function definition. For example:

```
CREATE OR REPLACE FUNCTION testfunc(INTEGER) RETURNS INTEGER AS '
    ....
end;
' LANGUAGE 'plpgsql';
```

While running `psql`, you can load or reload such a function definition file with

```
    \i filename.sql
```

and then immediately issue SQL commands to test the function.

Another good way to develop in PL/pgSQL is using PostgreSQL's GUI tool: PgAccess. It does some nice things for you, like escaping single-quotes, and making it easy to recreate and debug functions.

23.2. Structure of PL/pgSQL

PL/pgSQL is a *block structured* language. The complete text of a function definition must be a *block*. A block is defined as:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END;
```

Any *statement* in the statement section of a block can be a *sub-block*. Sub-blocks can be used for logical grouping or to localize variables to a small group of statements.

The variables declared in the declarations section preceding a block are initialized to their default values every time the block is entered, not only once per function call. For example:

```
CREATE FUNCTION somefunc() RETURNS INTEGER AS '
DECLARE
   quantity INTEGER := 30;
BEGIN
   RAISE NOTICE ''Quantity here is %'',quantity;  -- Quantity here is
 30
   quantity := 50;
   --
   -- Create a sub-block
   --
   DECLARE
      quantity INTEGER := 80;
   BEGIN
      RAISE NOTICE ''Quantity here is %'',quantity;  -- Quantity here
 is 80
   END;

   RAISE NOTICE ''Quantity here is %'',quantity;  -- Quantity here is
 50

   RETURN quantity;
END;
```

```
' LANGUAGE 'plpgsql';
```

It is important not to confuse the use of BEGIN/END for grouping statements in PL/pgSQL with the database commands for transaction control. PL/pgSQL's BEGIN/END are only for grouping; they do not start or end a transaction. Functions and trigger procedures are always executed within a transaction established by an outer query --- they cannot start or commit transactions, since PostgreSQL does not have nested transactions.

23.2.1. Lexical Details

Each statement and declaration within a block is terminated by a semicolon.

All keywords and identifiers can be written in mixed upper- and lower-case. Identifiers are implicitly converted to lower-case unless double-quoted.

There are two types of comments in PL/pgSQL. A double dash `--` starts a comment that extends to the end of the line. A `/*` starts a block comment that extends to the next occurrence of `*/`. Block comments cannot be nested, but double dash comments can be enclosed into a block comment and a double dash can hide the block comment delimiters `/*` and `*/`.

23.3. Declarations

All variables, rows and records used in a block must be declared in the declarations section of the block. (The only exception is that the loop variable of a FOR loop iterating over a range of integer values is automatically declared as an integer variable.)

PL/pgSQL variables can have any SQL data type, such as `INTEGER`, `VARCHAR` and `CHAR`.

Here are some examples of variable declarations:

```
user_id INTEGER;
quantity NUMERIC(5);
url VARCHAR;
```

The general syntax of a variable declaration is:

```
name [ CONSTANT ] type [ NOT NULL ] [ { DEFAULT | := } expression ];
```

The DEFAULT clause, if given, specifies the initial value assigned to the variable when the block is entered. If the DEFAULT clause is not given then the variable is initialized to the SQL NULL value.

The CONSTANT option prevents the variable from being assigned to, so that its value remains constant for the duration of the block. If NOT NULL is specified, an assignment of a NULL value results in a runtime error. All variables declared as NOT NULL must have a non-NULL default value specified.

The default value is evaluated every time the block is entered. So, for example, assigning '`now`' to a variable of type `timestamp` causes the variable to have the time of the current function call, not when the function was precompiled.

Examples:

```
quantity INTEGER DEFAULT 32;
url varchar := ''http://mysite.com'';
user_id CONSTANT INTEGER := 10;
```

23.3.1. Aliases for Function Parameters

```
name ALIAS FOR $n;
```

Parameters passed to functions are named with the identifiers `$1`, `$2`, etc. Optionally, aliases can be declared for `$n` parameter names for increased readability. Either the alias or the numeric identifier can then be used to refer to the parameter value. Some examples:

```
CREATE FUNCTION sales_tax(REAL) RETURNS REAL AS '
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    return subtotal * 0.06;
END;
' LANGUAGE 'plpgsql';


CREATE FUNCTION instr(VARCHAR,INTEGER) RETURNS INTEGER AS '
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- Some computations here
END;
' LANGUAGE 'plpgsql';
```

23.3.2. Rowtypes

```
name tablename%ROWTYPE;
```

A variable of a composite type is called a *row* variable (or *rowtype* variable). Such a variable can hold a whole row of a SELECT or FOR query result, so long as that query's column set matches the declared type of the variable. The individual fields of the row value are accessed using the usual dot notation, for example `rowvar.field`.

Presently, a row variable can only be declared using the `%ROWTYPE` notation; although one might expect a bare table name to work as a type declaration, it won't be accepted within PL/pgSQL functions.

Parameters to a function can be composite types (complete table rows). In that case, the corresponding identifier $n will be a row variable, and fields can be selected from it, for example `$1.user_id`.

Only the user-defined attributes of a table row are accessible in a rowtype variable, not OID or other system attributes (because the row could be from a view). The fields of the rowtype inherit the table's field size or precision for data types such as `char(n)`.

23.3.3. Records

```
name RECORD;
```

Record variables are similar to rowtype variables, but they have no predefined structure. They take on the actual row structure of the row they are assigned during a SELECT or FOR command. The substructure of a record variable can change each time it is assigned to. A consequence of this is that until a record variable is first assigned to, *it has no* substructure, and any attempt to access a field in it will draw a runtime error.

Note that RECORD is not a true datatype, only a placeholder. Thus, for example, one cannot declare a function returning RECORD.

23.3.4. Attributes

Using the %TYPE and %ROWTYPE attributes, you can declare variables with the same data type or structure as another database item (e.g: a table field).

variable%TYPE

%TYPE provides the data type of a variable or database column. You can use this to declare variables that will hold database values. For example, let's say you have a column named user_id in your users table. To declare a variable with the same data type as users.user_id you write:

```
user_id   users.user_id%TYPE;
```

By using %TYPE you don't need to know the data type of the structure you are referencing, and most important, if the data type of the referenced item changes in the future (e.g: you change your table definition of user_id from INTEGER to REAL), you may not need to change your function definition.

table%ROWTYPE

%ROWTYPE provides the composite data type corresponding to a whole row of the specified table. *table* must be an existing table or view name of the database.

```
DECLARE
    users_rec users%ROWTYPE;
    user_id users.user_id%TYPE;
BEGIN
    user_id := users_rec.user_id;
    ...
```

```
CREATE FUNCTION does_view_exist(INTEGER) RETURNS bool AS '
   DECLARE
        key ALIAS FOR $1;
        table_data cs_materialized_views%ROWTYPE;
   BEGIN
        SELECT INTO table_data * FROM cs_materialized_views
               WHERE sort_key=key;

        IF NOT FOUND THEN
           RETURN false;
        END IF;
        RETURN true;
   END;
' LANGUAGE 'plpgsql';
```

23.3.5. RENAME

RENAME *oldname* TO *newname*;

Using the RENAME declaration you can change the name of a variable, record or row. This is primarily useful if NEW or OLD should be referenced by another name inside a trigger procedure. See also ALIAS.

Examples:

```
RENAME id TO user_id;
RENAME this_var TO that_var;
```

Note

RENAME appears to be broken as of PostgreSQL 7.2. Fixing this is of low priority, since ALIAS covers most of the practical uses of RENAME.

23.4. Expressions

All expressions used in PL/pgSQL statements are processed using the server's regular SQL executor. Expressions that appear to contain constants may in fact require run-time evaluation (e.g. `'now'` for the `timestamp` type) so it is impossible for the PL/pgSQL parser to identify real constant values other than the NULL keyword. All expressions are evaluated internally by executing a query

```
SELECT expression
```

using the SPI manager. In the expression, occurrences of PL/pgSQL variable identifiers are replaced by parameters and the actual values from the variables are passed to the executor in the parameter array. This allows the query plan for the SELECT to be prepared just once and then re-used for subsequent evaluations.

The evaluation done by the PostgreSQL main parser has some side effects on the interpretation of constant values. In detail there is a difference between what these two functions do:

```
CREATE FUNCTION logfunc1 (TEXT) RETURNS TIMESTAMP AS '
    DECLARE
        logtxt ALIAS FOR $1;
    BEGIN
        INSERT INTO logtable VALUES (logtxt, ''now'');
        RETURN ''now'';
    END;
' LANGUAGE 'plpgsql';
```

and

```
CREATE FUNCTION logfunc2 (TEXT) RETURNS TIMESTAMP AS '
    DECLARE
        logtxt ALIAS FOR $1;
        curtime timestamp;
    BEGIN
        curtime := ''now'';
        INSERT INTO logtable VALUES (logtxt, curtime);
        RETURN curtime;
    END;
' LANGUAGE 'plpgsql';
```

In the case of `logfunc1()`, the PostgreSQL main parser knows when preparing the plan for the INSERT, that the string `'now'` should be interpreted as `timestamp` because the target field of `logtable` is of that type. Thus, it will make a constant from it at this time and this constant value is then used in all invocations of `logfunc1()` during the lifetime of the backend. Needless to say that this isn't what the programmer wanted.

In the case of `logfunc2()`, the PostgreSQL main parser does not know what type `'now'` should become and therefore it returns a data value of type `text` containing the string `'now'`. During the ensuing

assignment to the local variable `curtime`, the PL/pgSQL interpreter casts this string to the `timestamp` type by calling the `text_out()` and `timestamp_in()` functions for the conversion. So, the computed timestamp is updated on each execution as the programmer expects.

The mutable nature of record variables presents a problem in this connection. When fields of a record variable are used in expressions or statements, the data types of the fields must not change between calls of one and the same expression, since the expression will be planned using the datatype that is present when the expression is first reached. Keep this in mind when writing trigger procedures that handle events for more than one table. (EXECUTE can be used to get around this problem when necessary.)

23.5. Basic Statements

In this section and the following ones, we describe all the statement types that are explicitly understood by PL/pgSQL. Anything not recognized as one of these statement types is presumed to be an SQL query, and is sent to the main database engine to execute (after substitution for any PL/pgSQL variables used in the statement). Thus, for example, SQL `INSERT`, `UPDATE`, and `DELETE` commands may be considered to be statements of PL/pgSQL. But they are not specifically listed here.

23.5.1. Assignment

An assignment of a value to a variable or row/record field is written as:

```
identifier := expression;
```

As explained above, the expression in such a statement is evaluated by means of an SQL `SELECT` command sent to the main database engine. The expression must yield a single value.

If the expression's result data type doesn't match the variable's data type, or the variable has a specific size/precision (as for `char(20)`), the result value will be implicitly converted by the PL/pgSQL interpreter using the result type's output-function and the variable type's input-function. Note that this could potentially result in runtime errors generated by the input function, if the string form of the result value is not acceptable to the input function.

Examples:

```
user_id := 20;
tax := subtotal * 0.06;
```

23.5.2. SELECT INTO

The result of a SELECT command yielding multiple columns (but only one row) can be assigned to a record variable, rowtype variable, or list of scalar variables. This is done by:

```
SELECT INTO target expressions FROM ...;
```

where *`target`* can be a record variable, a row variable, or a comma-separated list of simple variables and record/row fields. Note that this is quite different from PostgreSQL's normal interpretation of SELECT INTO, which is that the INTO target is a newly created table. (If you want to create a table from a SELECT result inside a PL/pgSQL function, use the syntax `CREATE TABLE ... AS SELECT`.)

If a row or a variable list is used as target, the selected values must exactly match the structure of the target(s), or a runtime error occurs. When a record variable is the target, it automatically configures itself to the rowtype of the query result columns.

Except for the INTO clause, the SELECT statement is the same as a normal SQL SELECT query and can use the full power of SELECT.

If the SELECT query returns zero rows, NULLs are assigned to the target(s). If the SELECT query returns multiple rows, the first row is assigned to the target(s) and the rest are discarded. (Note that “the first row” is not well-defined unless you've used ORDER BY.)

At present, the INTO clause can appear almost anywhere in the SELECT query, but it is recommended to place it immediately after the SELECT keyword as depicted above. Future versions of PL/pgSQL may be less forgiving about placement of the INTO clause.

There is a special variable named FOUND of type `boolean` that can be used immediately after a SELECT INTO to check if an assignment had success (that is, at least one row was returned by the SELECT). For example,

```
SELECT INTO myrec * FROM EMP WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION ''employee % not found'', myname;
END IF;
```

Alternatively, you can use the IS NULL (or ISNULL) conditional to test for NULLity of a RECORD/ROW result. Note that there is no way to tell whether any additional rows might have been discarded.

```
DECLARE
    users_rec RECORD;
    full_name varchar;
BEGIN
    SELECT INTO users_rec * FROM users WHERE user_id=3;

    IF users_rec.homepage IS NULL THEN
        -- user entered no homepage, return "http://"

        RETURN ''http://'';
    END IF;
END;
```

23.5.3. Executing an expression or query with no result

Sometimes one wishes to evaluate an expression or query but discard the result (typically because one is calling a function that has useful side-effects but no useful result value). To do this in PL/pgSQL, use the PERFORM statement:

```
PERFORM query;
```

This executes a `SELECT` *`query`* and discards the result. PL/pgSQL variables are substituted into the query as usual.

Note

One might expect that SELECT with no INTO clause would accomplish this result, but at present the only accepted way to do it is PERFORM.

An example:

```
PERFORM create_mv(''cs_session_page_requests_mv'',''
```

```
    SELECT   session_id, page_id, count(*) AS n_hits,
             sum(dwell_time) AS dwell_time, count(dwell_time) AS
 dwell_count
    FROM     cs_fact_table
    GROUP BY session_id, page_id '');
```

23.5.4. Executing dynamic queries

Oftentimes you will want to generate dynamic queries inside your PL/pgSQL functions, that is, queries that will involve different tables or different datatypes each time they are executed. PL/pgSQL's normal attempts to cache plans for queries will not work in such scenarios. To handle this sort of problem, the EXECUTE statement is provided:

```
EXECUTE query-string;
```

where *query-string* is an expression yielding a string (of type `text`) containing the *query* to be executed. This string is fed literally to the SQL engine.

Note in particular that no substitution of PL/pgSQL variables is done on the query string. The values of variables must be inserted into the query string as it is constructed.

When working with dynamic queries you will have to face escaping of single quotes in PL/pgSQL. Please refer to the table in Section 23.11, “Porting from Oracle PL/SQL” for a detailed explanation that will save you some effort.

Unlike all other queries in PL/pgSQL, a *query* run by an EXECUTE statement is not prepared and saved just once during the life of the server. Instead, the *query* is prepared each time the statement is run. The *query-string* can be dynamically created within the procedure to perform actions on variable tables and fields.

The results from SELECT queries are discarded by EXECUTE, and SELECT INTO is not currently supported within EXECUTE. So, the only way to extract a result from a dynamically-created SELECT is to use the FOR-IN-EXECUTE form described later.

An example:

```
EXECUTE ''UPDATE tbl SET ''
        || quote_ident(fieldname)
        || '' = ''
        || quote_literal(newvalue)
        || '' WHERE ...'';
```

This example shows use of the functions `quote_ident(TEXT)` and `quote_literal(TEXT)`. Variables containing field and table identifiers should be passed to function `quote_ident()`. Variables containing literal elements of the dynamic query string should be passed to `quote_literal()`. Both take the appropriate steps to return the input text enclosed in single or double quotes and with any embedded special characters properly escaped.

Here is a much larger example of a dynamic query and EXECUTE:

```
CREATE FUNCTION cs_update_referrer_type_proc() RETURNS INTEGER AS '
DECLARE
    referrer_keys RECORD;  -- Declare a generic record to be used in a
 FOR
    a_output varchar(4000);
```

```
BEGIN
    a_output := ''CREATE FUNCTION
 cs_find_referrer_type(varchar,varchar,varchar)
                 RETURNS VARCHAR AS ''''
                    DECLARE
                        v_host ALIAS FOR $1;
                        v_domain ALIAS FOR $2;
                        v_url ALIAS FOR $3;
                    BEGIN '';

    --
    -- Notice how we scan through the results of a query in a FOR loop
    -- using the FOR <record> construct.
    --

    FOR referrer_keys IN SELECT * FROM cs_referrer_keys ORDER BY
 try_order LOOP
        a_output := a_output || '' IF v_'' || referrer_keys.kind || ''
 LIKE ''''''''''
                || referrer_keys.key_string || '''''''''' THEN RETURN
 ''''''
                || referrer_keys.referrer_type || ''''''; END IF;'';
    END LOOP;

    a_output := a_output || '' RETURN NULL; END; '''' LANGUAGE
 ''''plpgsql'''';'';

    -- This works because we are not substituting any variables
    -- Otherwise it would fail. Look at PERFORM for another way to run
 functions

    EXECUTE a_output;
END;
' LANGUAGE 'plpgsql';
```

23.5.5. Obtaining result status

```
GET DIAGNOSTICS variable = item [ , ... ] ;
```

This command allows retrieval of system status indicators. Each *item* is a keyword identifying a state value to be assigned to the specified variable (which should be of the right data type to receive it). The currently available status items are ROW_COUNT, the number of rows processed by the last SQL query sent down to the SQL engine; and RESULT_OID, the Oid of the last row inserted by the most recent SQL query. Note that RESULT_OID is only useful after an INSERT query.

23.6. Control Structures

Control structures are probably the most useful (and important) part of PL/pgSQL. With PL/pgSQL's control structures, you can manipulate PostgreSQL data in a very flexible and powerful way.

23.6.1. Returning from a function

```
RETURN expression;
```

The function terminates and the value of *expression* will be returned to the upper executor. The expression's result will be automatically casted into the function's return type as described for assignments.

The return value of a function cannot be left undefined. If control reaches the end of the top-level block of the function without hitting a RETURN statement, a runtime error will occur.

23.6.2. Conditionals

IF statements let you execute commands based on certain conditions. PL/pgSQL has four forms of IF: IF-THEN, IF-THEN-ELSE, IF-THEN-ELSE IF, and IF-THEN-ELSIF-THEN-ELSE.

23.6.2.1. IF-THEN

```
IF boolean-expression THEN
    statements
END IF;
```

IF-THEN statements are the simplest form of IF. The statements between THEN and END IF will be executed if the condition is true. Otherwise, they are skipped.

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

23.6.2.2. IF-THEN-ELSE

```
IF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```

IF-THEN-ELSE statements add to IF-THEN by letting you specify an alternative set of statements that should be executed if the condition evaluates to FALSE.

```
IF parentid IS NULL or parentid = ''''
THEN
    return fullname;
ELSE
    return hp_true_filename(parentid) || ''/'' || fullname;
END IF;


IF v_count > 0 THEN
    INSERT INTO users_count(count) VALUES(v_count);
    return ''t'';
ELSE
    return ''f'';
END IF;
```

23.6.2.3. IF-THEN-ELSE IF

IF statements can be nested, as in the following example:

```
IF demo_row.sex = ''m'' THEN
```

```
  pretty_sex := ''man'';
ELSE
  IF demo_row.sex = ''f'' THEN
    pretty_sex := ''woman'';
  END IF;
END IF;
```

When you use this form, you are actually nesting an IF statement inside the ELSE part of an outer IF statement. Thus you need one END IF statement for each nested IF and one for the parent IF-ELSE. This is workable but grows tedious when there are many alternatives to be checked.

23.6.2.4. IF-THEN-ELSIF-ELSE

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```

IF-THEN-ELSIF-ELSE provides a more convenient method of checking many alternatives in one statement. Formally it is equivalent to nested IF-THEN-ELSE-IF-THEN commands, but only one END IF is needed.

Here is an example:

```
IF number = 0 THEN
    result := ''zero'';
ELSIF number > 0 THEN
    result := ''positive'';
ELSIF number < 0 THEN
    result := ''negative'';
ELSE
    -- hmm, the only other possibility is that number IS NULL
    result := ''NULL'';
END IF;
```

The final ELSE section is optional.

23.6.3. Simple Loops

With the LOOP, EXIT, WHILE and FOR statements, you can arrange for your PL/pgSQL function to repeat a series of commands.

23.6.3.1. LOOP

```
[<<label>>]
LOOP
    statements
END LOOP;
```

LOOP defines an unconditional loop that is repeated indefinitely until terminated by an EXIT or RETURN statement. The optional label can be used by EXIT statements in nested loops to specify which level of nesting should be terminated.

23.6.3.2. EXIT

```
EXIT [ label ] [ WHEN expression ];
```

If no *label* is given, the innermost loop is terminated and the statement following END LOOP is executed next. If *label* is given, it must be the label of the current or some outer level of nested loop or block. Then the named loop or block is terminated and control continues with the statement after the loop's/block's corresponding END.

If WHEN is present, loop exit occurs only if the specified condition is true, otherwise control passes to the statement after EXIT.

Examples:

```
LOOP
    -- some computations
    IF count > 0 THEN
        EXIT;  -- exit loop
    END IF;
END LOOP;

LOOP
    -- some computations
    EXIT WHEN count > 0;
END LOOP;

BEGIN
    -- some computations
    IF stocks > 100000 THEN
        EXIT;  -- illegal. Can't use EXIT outside of a LOOP
    END IF;
END;
```

23.6.3.3. WHILE

```
[<<label>>]
WHILE expression LOOP
    statements
END LOOP;
```

The WHILE statement repeats a sequence of statements so long as the condition expression evaluates to true. The condition is checked just before each entry to the loop body.

For example:

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- some computations here
END LOOP;

WHILE NOT boolean_expression LOOP
    -- some computations here
```

```
END LOOP;
```

23.6.3.4. FOR (integer for-loop)

```
[<<label>>]
FOR name IN [ REVERSE ] expression .. expression LOOP
    statements
END LOOP;
```

This form of FOR creates a loop that iterates over a range of integer values. The variable *name* is automatically defined as type integer and exists only inside the loop. The two expressions giving the lower and upper bound of the range are evaluated once when entering the loop. The iteration step is normally 1, but is -1 when REVERSE is specified.

Some examples of integer FOR loops:

```
FOR i IN 1..10 LOOP
  -- some expressions here

    RAISE NOTICE ''i is %'',i;
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- some expressions here
END LOOP;
```

23.6.4. Looping Through Query Results

Using a different type of FOR loop, you can iterate through the results of a query and manipulate that data accordingly. The syntax is:

```
[<<label>>]
FOR record | row IN select_query LOOP
    statements
END LOOP;
```

The record or row variable is successively assigned all the rows resulting from the SELECT query and the loop body is executed for each row. Here is an example:

```
CREATE FUNCTION cs_refresh_mviews () RETURNS INTEGER AS '
DECLARE
     mviews RECORD;
BEGIN
     PERFORM cs_log(''Refreshing materialized views...'');

     FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY
 sort_key LOOP

         -- Now "mviews" has one record from cs_materialized_views

         PERFORM cs_log(''Refreshing materialized view '' ||
 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(''Done refreshing materialized views.'');
     RETURN 1;
end;
' LANGUAGE 'plpgsql';
```

If the loop is terminated by an EXIT statement, the last assigned row value is still accessible after the loop.

The FOR-IN-EXECUTE statement is another way to iterate over records:

```
[<<label>>]
FOR record | row IN EXECUTE text_expression LOOP
    statements
END LOOP;
```

This is like the previous form, except that the source SELECT statement is specified as a string expression, which is evaluated and re-planned on each entry to the FOR loop. This allows the programmer to choose the speed of a pre-planned query or the flexibility of a dynamic query, just as with a plain EXECUTE statement.

Note

The PL/pgSQL parser presently distinguishes the two kinds of FOR loops (integer or record-returning) by checking whether the target variable mentioned just after FOR has been declared as a record/row variable. If not, it's presumed to be an integer FOR loop. This can cause rather unintuitive error messages when the true problem is, say, that one has misspelled the FOR variable name.

23.7. Cursors

Rather than executing a whole query at once, it is possible to set up a *cursor* that encapsulates the query, and then read the query result a few rows at a time. One reason for doing this is to avoid memory overrun when the result contains a large number of rows. (However, PL/pgSQL users don't normally need to worry about that, since FOR loops automatically use a cursor internally to avoid memory problems.) A more interesting possibility is that a function can return a reference to a cursor that it has set up, allowing the caller to read the rows. This provides one way of returning a rowset from a function.

23.7.1. Declaring Cursor Variables

All access to cursors in PL/pgSQL goes through cursor variables, which are always of the special datatype `refcursor`. One way to create a cursor variable is just to declare it as a variable of type `refcursor`. Another way is to use the cursor declaration syntax, which in general is:

```
name CURSOR [ ( arguments ) ] FOR select_query ;
```

(`FOR` may be replaced by `IS` for Oracle compatibility.) *`arguments`*, if any, are a comma-separated list of *`name datatype`* pairs that define names to be replaced by parameter values in the given query. The actual values to substitute for these names will be specified later, when the cursor is opened.

Some examples:

```
DECLARE
    curs1 refcursor;
```

```
curs2 CURSOR FOR SELECT * from tenk1;
curs3 CURSOR (key int) IS SELECT * from tenk1 where unique1 = key;
```

All three of these variables have the datatype `refcursor`, but the first may be used with any query, while the second has a fully specified query already *bound* to it, and the last has a parameterized query bound to it. (`key` will be replaced by an integer parameter value when the cursor is opened.) The variable `curs1` is said to be *unbound* since it is not bound to any particular query.

23.7.2. Opening Cursors

Before a cursor can be used to retrieve rows, it must be *opened*. (This is the equivalent action to the SQL command `DECLARE CURSOR`.) PL/pgSQL has four forms of the OPEN statement, two of which are for use with unbound cursor variables and the other two for use with bound cursor variables.

23.7.2.1. OPEN FOR SELECT

```
OPEN unbound-cursor FOR SELECT ...;
```

The cursor variable is opened and given the specified query to execute. The cursor cannot be open already, and it must have been declared as an unbound cursor (that is, as a simple `refcursor` variable). The SELECT query is treated in the same way as other SELECTs in PL/pgSQL: PL/pgSQL variable names are substituted for, and the query plan is cached for possible re-use.

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

23.7.2.2. OPEN FOR EXECUTE

```
OPEN unbound-cursor FOR EXECUTE query-string;
```

The cursor variable is opened and given the specified query to execute. The cursor cannot be open already, and it must have been declared as an unbound cursor (that is, as a simple `refcursor` variable). The query is specified as a string expression in the same way as for the EXECUTE command. As usual, this gives flexibility for the query to vary from one run to the next.

```
OPEN curs1 FOR EXECUTE ''SELECT * FROM '' || quote_ident($1);
```

23.7.2.3. OPENing a bound cursor

```
OPEN bound-cursor [ ( argument_values ) ];
```

This form of OPEN is used to open a cursor variable whose query was bound to it when it was declared. The cursor cannot be open already. A list of actual argument value expressions must appear if and only if the cursor was declared to take arguments. These values will be substituted into the query. The query plan for a bound cursor is always considered cacheable --- there is no equivalent of EXECUTE in this case.

```
OPEN curs2;
OPEN curs3(42);
```

23.7.3. Using Cursors

Once a cursor has been opened, it can be manipulated with the statements described here.

These manipulations need not occur in the same function that opened the cursor to begin with. You can return a `refcursor` value out of a function and let the caller operate on the cursor. (Internally, a `refcursor` value is simply the string name of a Portal containing the active query for the cursor. This

name can be passed around, assigned to other `refcursor` variables, and so on, without disturbing the Portal.)

All Portals are implicitly closed at end of transaction. Therefore a `refcursor` value is useful to reference an open cursor only until the end of the transaction.

23.7.3.1. FETCH

```
FETCH cursor INTO target;
```

FETCH retrieves the next row from the cursor into a target, which may be a row variable, a record variable, or a comma-separated list of simple variables, just as for SELECT INTO. As with SELECT INTO, the special variable FOUND may be checked to see whether a row was obtained or not.

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo,bar,baz;
```

23.7.3.2. CLOSE

```
CLOSE cursor;
```

CLOSE closes the Portal underlying an open cursor. This can be used to release resources earlier than end of transaction, or to free up the cursor variable to be opened again.

```
CLOSE curs1;
```

23.8. Errors and Messages

Use the RAISE statement to report messages and raise errors.

```
RAISE level 'format' [, variable [...]];
```

Possible levels are DEBUG (write the message into the postmaster log), NOTICE (write the message into the postmaster log and forward it to the client application) and EXCEPTION (raise an error, aborting the transaction).

Inside the format string, `%` is replaced by the next optional argument's external representation. Write `%%` to emit a literal `%`. Note that the optional arguments must presently be simple variables, not expressions, and the format must be a simple string literal.

Examples:

```
RAISE NOTICE ''Calling cs_create_job(%)'',v_job_id;
```

In this example, the value of v_job_id will replace the % in the string.

```
RAISE EXCEPTION ''Inexistent ID --> %'',user_id;
```

This will abort the transaction with the given error message.

23.8.1. Exceptions

PostgreSQL does not have a very smart exception handling model. Whenever the parser, planner/optimizer or executor decide that a statement cannot be processed any longer, the whole transaction gets aborted and the system jumps back into the main loop to get the next query from the client application.

It is possible to hook into the error mechanism to notice that this happens. But currently it is impossible to tell what really caused the abort (input/output conversion error, floating-point error, parse error). And it is possible that the database backend is in an inconsistent state at this point so returning to the upper executor or issuing more commands might corrupt the whole database.

Thus, the only thing PL/pgSQL currently does when it encounters an abort during execution of a function or trigger procedure is to write some additional NOTICE level log messages telling in which function and where (line number and type of statement) this happened. The error always stops execution of the function.

23.9. Trigger Procedures

PL/pgSQL can be used to define trigger procedures. A trigger procedure is created with the `CREATE FUNCTION` command as a function with no arguments and a return type of `OPAQUE`. Note that the function must be declared with no arguments even if it expects to receive arguments specified in `CREATE TRIGGER` --- trigger arguments are passed via `TG_ARGV`, as described below.

When a PL/pgSQL function is called as a trigger, several special variables are created automatically in the top-level block. They are:

`NEW`

Data type `RECORD`; variable holding the new database row for INSERT/UPDATE operations in ROW level triggers.

`OLD`

Data type `RECORD`; variable holding the old database row for UPDATE/DELETE operations in ROW level triggers.

`TG_NAME`

Data type `name`; variable that contains the name of the trigger actually fired.

`TG_WHEN`

Data type `text`; a string of either `BEFORE` or `AFTER` depending on the trigger's definition.

`TG_LEVEL`

Data type `text`; a string of either `ROW` or `STATEMENT` depending on the trigger's definition.

`TG_OP`

Data type `text`; a string of `INSERT`, `UPDATE` or `DELETE` telling for which operation the trigger is fired.

`TG_RELID`

Data type `oid`; the object ID of the table that caused the trigger invocation.

`TG_RELNAME`

Data type `name`; the name of the table that caused the trigger invocation.

`TG_NARGS`

Data type `integer`; the number of arguments given to the trigger procedure in the `CREATE TRIGGER` statement.

`TG_ARGV[]`

Data type array of `text`; the arguments from the `CREATE TRIGGER` statement. The index counts from 0 and can be given as an expression. Invalid indices (< 0 or >= tg_nargs) result in a NULL value.

A trigger function must return either NULL or a record/row value having exactly the structure of the table the trigger was fired for. Triggers fired BEFORE may return NULL to signal the trigger manager to skip the rest of the operation for this row (ie, subsequent triggers are not fired, and the INSERT/UPDATE/DELETE does not occur for this row). If a non-NULL value is returned then the operation proceeds with that row value. Note that returning a row value different from the original value of NEW alters the row that will be inserted or updated. It is possible to replace single values directly in NEW and return that, or to build a complete new record/row to return.

The return value of a trigger fired AFTER is ignored; it may as well always return a NULL value. But an AFTER trigger can still abort the operation by raising an error.

Example 23.1. A PL/pgSQL Trigger Procedure Example

This example trigger ensures that any time a row is inserted or updated in the table, the current user name and time are stamped into the row. And it ensures that an employee's name is given and that the salary is a positive value.

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp () RETURNS OPAQUE AS '
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname ISNULL THEN
            RAISE EXCEPTION ''empname cannot be NULL value'';
        END IF;
        IF NEW.salary ISNULL THEN
            RAISE EXCEPTION ''% cannot have NULL salary'',
 NEW.empname;
        END IF;

        -- Who works for us when she must pay for?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION ''% cannot have a negative salary'',
 NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := ''now'';
        NEW.last_user := current_user;
        RETURN NEW;
    END;
' LANGUAGE 'plpgsql';

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

23.10. Examples

Here are only a few functions to demonstrate how easy it is to write PL/pgSQL functions. For more complex examples the programmer might look at the regression test for PL/pgSQL.

One painful detail in writing functions in PL/pgSQL is the handling of single quotes. The function's source text in `CREATE FUNCTION` must be a literal string. Single quotes inside of literal strings must be either doubled or quoted with a backslash. We are still looking for an elegant alternative. In the meantime, doubling the single quotes as in the examples below should be used. Any solution for this in future versions of PostgreSQL will be forward compatible.

For a detailed explanation and examples of how to escape single quotes in different situations, please see Section 23.11.1.1, “Quote Me on That: Escaping Single Quotes”.

Example 23.2. A Simple PL/pgSQL Function to Increment an Integer

The following two PL/pgSQL functions are identical to their counterparts from the C language function discussion. This function receives an `integer` and increments it by one, returning the incremented value.

```
CREATE FUNCTION add_one (integer) RETURNS INTEGER AS '
    BEGIN
        RETURN $1 + 1;
    END;
' LANGUAGE 'plpgsql';
```

Example 23.3. A Simple PL/pgSQL Function to Concatenate Text

This function receives two `text` parameters and returns the result of concatenating them.

```
CREATE FUNCTION concat_text (TEXT, TEXT) RETURNS TEXT AS '
    BEGIN
        RETURN $1 || $2;
    END;
' LANGUAGE 'plpgsql';
```

Example 23.4. A PL/pgSQL Function on Composite Type

In this example, we take `EMP` (a table) and an `integer` as arguments to our function, which returns a `boolean`. If the `salary` field of the `EMP` table is `NULL`, we return `f`. Otherwise we compare with that field with the `integer` passed to the function and return the `boolean` result of the comparison (t or f). This is the PL/pgSQL equivalent to the example from the C functions.

```
CREATE FUNCTION c_overpaid (EMP, INTEGER) RETURNS BOOLEAN AS '
    DECLARE
        emprec ALIAS FOR $1;
        sallim ALIAS FOR $2;
    BEGIN
        IF emprec.salary ISNULL THEN
            RETURN ''f'';
        END IF;
        RETURN emprec.salary > sallim;
    END;
' LANGUAGE 'plpgsql';
```

23.11. Porting from Oracle PL/SQL

Roberto Mello <rmello@fslc.usu.edu>

Author

Roberto Mello (<rmello@fslc.usu.edu>)

This section explains differences between Oracle's PL/SQL and PostgreSQL's PL/pgSQL languages in the hopes of helping developers port applications from Oracle to PostgreSQL. Most of the code here is from the ArsDigita[1] Clickstream module[2] that I ported to PostgreSQL when I took an internship with OpenForce Inc.[3] in the Summer of 2000.

PL/pgSQL is similar to PL/SQL in many aspects. It is a block structured, imperative language (all variables have to be declared). PL/SQL has many more features than its PostgreSQL counterpart, but PL/pgSQL allows for a great deal of functionality and it is being improved constantly.

23.11.1. Main Differences

Some things you should keep in mind when porting from Oracle to PostgreSQL:

- No default parameters in PostgreSQL.
- You can overload functions in PostgreSQL. This is often used to work around the lack of default parameters.
- Assignments, loops and conditionals are similar.
- No need for cursors in PostgreSQL, just put the query in the FOR statement (see example below)
- In PostgreSQL you *need* to escape single quotes. See Section 23.11.1.1, “Quote Me on That: Escaping Single Quotes”.

23.11.1.1. Quote Me on That: Escaping Single Quotes

In PostgreSQL you need to escape single quotes inside your function definition. This can lead to quite amusing code at times, especially if you are creating a function that generates other function(s), as in Example 23.6, “ A Function that Creates Another Function ”. One thing to keep in mind when escaping lots of single quotes is that, except for the beginning/ending quotes, all the others will come in even quantity.

Table 23.1, “Single Quotes Escaping Chart” gives the scoop. (You'll love this little chart.)

Table 23.1. Single Quotes Escaping Chart

No. of Quotes	Usage	Example	Result
1	To begin/terminate function bodies	`CREATE FUNCTION foo() RETURNS INTEGER AS '...' LANGUAGE 'plpgsql';`	as is

[1] http://www.arsdigita.com
[2] http://www.arsdigita.com/asj/clickstream
[3] http://www.openforce.net

No. of Quotes	Usage	Example	Result
2	In assignments, SELECTs, to delimit strings, etc.	`a_output := ''Blah''; SELECT * FROM users WHERE f_name=''foobar'';`	`SELECT * FROM users WHERE f_name='foobar';`
4	When you need two single quotes in your resulting string without terminating that string.	`a_output := a_output || '' AND name LIKE ''''foobar'''' AND ...''`	`AND name LIKE 'foobar' AND ...`
6	When you want double quotes in your resulting string *and* terminate that string.	`a_output := a_output || '' AND name LIKE ''''foobar''''''`	`AND name LIKE 'foobar'`
10	When you want two single quotes in the resulting string (which accounts for 8 quotes) *and* terminate that string (2 more). You will probably only need that if you were using a function to generate other functions (like in Example 23.6, " A Function that Creates Another Function ").	`a_output := a_output || '' if v_'' || referrer_keys.kind || '' like '''''''''' || referrer_keys.key_string || '''''''''' then return '''''' || referrer_keys.referrer_type || ''''''; end if;'';`	`if v_<...> like ''<...>'' then return ''<...>''; end if;`

23.11.2. Porting Functions

Example 23.5. A Simple Function

Here is an Oracle function:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name IN varchar,
 v_version IN varchar)
RETURN varchar IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
SHOW ERRORS;
```

Let's go through this function and see the differences to PL/pgSQL:

- PostgreSQL does not have named parameters. You have to explicitly alias them inside your function.
- Oracle can have `IN`, `OUT`, and `INOUT` parameters passed to functions. The `INOUT`, for example, means that the parameter will receive a value and return another. PostgreSQL only has "IN" parameters and functions can return only a single value.
- The `RETURN` key word in the function prototype (not the function body) becomes `RETURNS` in PostgreSQL.
- On PostgreSQL functions are created using single quotes as delimiters, so you have to escape single quotes inside your functions (which can be quite annoying at times; see Section 23.11.1.1, "Quote Me on That: Escaping Single Quotes").
- The `/show errors` command does not exist in PostgreSQL.

So let's see how this function would look when ported to PostgreSQL:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(VARCHAR, VARCHAR)
RETURNS VARCHAR AS '
DECLARE
    v_name ALIAS FOR $1;
    v_version ALIAS FOR $2;
BEGIN
    IF v_version IS NULL THEN
        return v_name;
    END IF;
    RETURN v_name || ''/'' || v_version;
END;
' LANGUAGE 'plpgsql';
```

Example 23.6. A Function that Creates Another Function

The following procedure grabs rows from a `SELECT` statement and builds a large function with the results in `IF` statements, for the sake of efficiency. Notice particularly the differences in cursors, `FOR` loops, and the need to escape single quotes in PostgreSQL.

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;

    a_output VARCHAR(4000);
BEGIN
    a_output := '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
        a_output := a_output || ' IF v_' || referrer_key.kind || '
 LIKE ''' ||
referrer_key.key_string || ''' THEN RETURN ''' ||
 referrer_key.referrer_type ||
'''; END IF;';
    END LOOP;
```

```
    a_output := a_output || ' RETURN NULL; END;';
    EXECUTE IMMEDIATE a_output;
END;
/
show errors
```

Here is how this function would end up in PostgreSQL:

```
CREATE FUNCTION cs_update_referrer_type_proc() RETURNS INTEGER AS '
DECLARE
    referrer_keys RECORD;  -- Declare a generic record to be used in a
 FOR
    a_output varchar(4000);
BEGIN
    a_output := ''CREATE FUNCTION
 cs_find_referrer_type(VARCHAR,VARCHAR,VARCHAR)
                  RETURNS VARCHAR AS ''''
                     DECLARE
                         v_host ALIAS FOR $1;
                         v_domain ALIAS FOR $2;
                         v_url ALIAS FOR $3;
                     BEGIN '';

    --
    -- Notice how we scan through the results of a query in a FOR loop
    -- using the FOR <record> construct.
    --

    FOR referrer_keys IN SELECT * FROM cs_referrer_keys ORDER BY
 try_order LOOP
        a_output := a_output || '' IF v_'' || referrer_keys.kind || ''
 LIKE ''''''''''
                || referrer_keys.key_string || '''''''''' THEN RETURN
 ''''''
                || referrer_keys.referrer_type || ''''''; END IF;'';
    END LOOP;

    a_output := a_output || '' RETURN NULL; END; '''' LANGUAGE
 ''''plpgsql'''';'';

    -- This works because we are not substituting any variables
    -- Otherwise it would fail. Look at PERFORM for another way to run
 functions

    EXECUTE a_output;
END;
' LANGUAGE 'plpgsql';
```

Example 23.7. A Procedure with a lot of String Manipulation and OUT Parameters

The following Oracle PL/SQL procedure is used to parse a URL and return several elements (host, path and query). It is an procedure because in PL/pgSQL functions only one value can be returned (see Section 23.11.3, “ Procedures ”). In PostgreSQL, one way to work around this is to split the procedure in three different functions: one to return the host, another for the path and another for the query.

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- This will be passed back
    v_path OUT VARCHAR,  -- This one too
    v_query OUT VARCHAR) -- And this one
is
    a_pos1 INTEGER;
    a_pos2 INTEGER;
begin
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//'); -- PostgreSQL doesn't have an instr
 function

    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;
```

Here is how this procedure could be translated for PostgreSQL:

```
CREATE OR REPLACE FUNCTION cs_parse_url_host(VARCHAR) RETURNS VARCHAR
 AS '
DECLARE
    v_url ALIAS FOR $1;
    v_host VARCHAR;
    v_path VARCHAR;
    a_pos1 INTEGER;
    a_pos2 INTEGER;
    a_pos3 INTEGER;
BEGIN
    v_host := NULL;
    a_pos1 := instr(v_url,''//'');
```

```
    IF a_pos1 = 0 THEN
        RETURN '''';  -- Return a blank
    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 v_host;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2 );
    RETURN v_host;
END;
' LANGUAGE 'plpgsql';
```

Note

PostgreSQL does not have an `instr` function, so you can work around it using a combination of other functions. I got tired of doing this and created my own `instr` functions that behave exactly like Oracle's (it makes life easier). See the Section 23.11.6, " Appendix " for the code.

23.11.3. Procedures

Oracle procedures give a little more flexibility to the developer because nothing needs to be explicitly returned, but it can be through the use of `INOUT` or `OUT` parameters.

An example:

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
    PRAGMA AUTONOMOUS_TRANSACTION;1
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;2

    SELECT count(*) INTO a_running_job_count
    FROM cs_jobs
    WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- free lock3
        raise_application_error(-20000, 'Unable to create a new job: a
 job is currently running.');
    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; -- don't worry if
 it already exists4
    END;
```

```
    COMMIT;
END;
/
show errors
```

Procedures like this can be easily converted into PostgreSQL functions returning an `INTEGER`. This procedure in particular is interesting because it can teach us some things:

1. There is no `pragma` statement in PostgreSQL.
2. If you do a `LOCK TABLE` in PL/pgSQL, the lock will not be released until the calling transaction is finished.
3. You also cannot have transactions in PL/pgSQL procedures. The entire function (and other functions called from therein) is executed in a transaction and PostgreSQL rolls back the results if something goes wrong. Therefore only one `BEGIN` statement is allowed.
4. The exception when would have to be replaced by an `IF` statement.

So let's see one of the ways we could port this procedure to PL/pgSQL:

```
CREATE OR REPLACE FUNCTION cs_create_job(INTEGER) RETURNS INTEGER AS '
DECLARE
    v_job_id ALIAS FOR $1;
    a_running_job_count INTEGER;
    a_num INTEGER;
    -- PRAGMA AUTONOMOUS_TRANSACTION;
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
        -- COMMIT; -- free lock
        RAISE EXCEPTION ''Unable to create a new job: a job is
 currently running.'';
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    SELECT count(*) into a_num
    FROM cs_jobs
    WHERE job_id=v_job_id;
    IF NOT FOUND THEN  -- If nothing was returned in the last query
        -- This job is not in the table so lets insert it.
        INSERT INTO cs_jobs(job_id, start_stamp) VALUES (v_job_id,
 sysdate());
        RETURN 1;
    ELSE
        RAISE NOTICE ''Job already running.'';1
    END IF;

    RETURN 0;
END;
' LANGUAGE 'plpgsql';
```

1 Notice how you can raise notices (or errors) in PL/pgSQL.

23.11.4. Packages

Note

I haven't done much with packages myself, so if there are mistakes here, please let me know.

Packages are a way Oracle gives you to encapsulate PL/SQL statements and functions into one entity, like Java classes, where you define methods and objects. You can access these objects/methods with a "." (dot). Here is an example of an Oracle package from ACS 4 (the ArsDigita Community System[4]):

```
CREATE OR REPLACE PACKAGE BODY acs
AS
  FUNCTION add_user (
    user_id     IN users.user_id%TYPE DEFAULT NULL,
    object_type     IN acs_objects.object_type%TYPE DEFAULT 'user',
    creation_date   IN acs_objects.creation_date%TYPE DEFAULT sysdate,
    creation_user   IN acs_objects.creation_user%TYPE DEFAULT NULL,
    creation_ip     IN acs_objects.creation_ip%TYPE DEFAULT NULL,
  ...
  ) RETURN users.user_id%TYPE
  IS
    v_user_id       users.user_id%TYPE;
    v_rel_id        membership_rels.rel_id%TYPE;
  BEGIN
    v_user_id := acs_user.new (user_id, object_type, creation_date,
                creation_user, creation_ip, email, ...
    RETURN v_user_id;
  END;
END acs;
/
show errors
```

We port this to PostgreSQL by creating the different objects of the Oracle package as functions with a standard naming convention. We have to pay attention to some other details, like the lack of default parameters in PostgreSQL functions. The above package would become something like this:

```
CREATE FUNCTION
 acs__add_user(INTEGER,INTEGER,VARCHAR,DATETIME,INTEGER,INTEGER,...)
RETURNS INTEGER AS '
DECLARE
    user_id ALIAS FOR $1;
    object_type ALIAS FOR $2;
    creation_date ALIAS FOR $3;
    creation_user ALIAS FOR $4;
    creation_ip ALIAS FOR $5;
    ...
    v_user_id users.user_id%TYPE;
    v_rel_id membership_rels.rel_id%TYPE;
BEGIN
    v_user_id :=
 acs_user__new(user_id,object_type,creation_date,creation_user,creation_ip, ...);
```

[4] http://www.arsdigita.com/doc/

```
    ...

    RETURN v_user_id;
END;
' LANGUAGE 'plpgsql';
```

23.11.5. Other Things to Watch For

23.11.5.1. EXECUTE

The PostgreSQL version of `EXECUTE` works nicely, but you have to remember to use `quote_literal(TEXT)` and `quote_string(TEXT)` as described in Section 23.5.4, "Executing dynamic queries". Constructs of the type `EXECUTE ''SELECT * from $1'';` will not work unless you use these functions.

23.11.5.2. Optimizing PL/pgSQL Functions

PostgreSQL gives you two function creation modifiers to optimize execution: `iscachable` (function always returns the same result when given the same arguments) and `isstrict` (function returns NULL if any argument is NULL). Consult the `CREATE FUNCTION` reference for details.

To make use of these optimization attributes, you have to use the `WITH` modifier in your `CREATE FUNCTION` statement. Something like:

```
CREATE FUNCTION foo(...) RETURNS INTEGER AS '
...
' LANGUAGE 'plpgsql'
WITH (isstrict, iscachable);
```

23.11.6. Appendix

23.11.6.1. Code for my `instr` functions

```
--
-- instr functions that mimic Oracle's counterpart
-- Syntax: instr(string1,string2,[n],[m]) where [] denotes optional
 params.
--
-- Searches string1 beginning at the nth character for the mth
-- occurrence of string2. If n is negative, search backwards. If m is
-- not passed, assume 1 (search starts at first character).
--
-- by Roberto Mello (rmello@fslc.usu.edu)
-- modified by Robert Gaszewski (graszew@poland.com)
-- Licensed under the GPL v2 or later.
--

CREATE FUNCTION instr(VARCHAR,VARCHAR) RETURNS INTEGER AS '
DECLARE
    pos integer;
BEGIN
    pos:= instr($1,$2,1);
    RETURN pos;
```

```
END;
' LANGUAGE 'plpgsql';


CREATE FUNCTION instr(VARCHAR,VARCHAR,INTEGER) RETURNS INTEGER AS '
DECLARE
    string ALIAS FOR $1;
    string_to_search ALIAS FOR $2;
    beg_index ALIAS FOR $3;
    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;
' LANGUAGE 'plpgsql';

--
-- Written by Robert Gaszewski (graszew@poland.com)
-- Licensed under the GPL v2 or later.
--
CREATE FUNCTION instr(VARCHAR,VARCHAR,INTEGER,INTEGER) RETURNS INTEGER
 AS '
DECLARE
    string ALIAS FOR $1;
    string_to_search ALIAS FOR $2;
```

```
    beg_index ALIAS FOR $3;
    occur_index ALIAS FOR $4;
    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';
```

Chapter 24. PL/Tcl - Tcl Procedural Language

PL/Tcl is a loadable procedural language for the PostgreSQL database system that enables the Tcl language to be used to write functions and trigger procedures.

This package was originally written by Jan Wieck.

24.1. Overview

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 backend or to gain OS-level access under the permissions of the PostgreSQL user ID, 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 procedures 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 mail. 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/Tk 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` script, for example `createlang pltcl` *`dbname`* or `createlang pltclu` *`dbname`*.

24.2. Description

24.2.1. PL/Tcl Functions and Arguments

To create a function in the PL/Tcl language, use the standard syntax

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS '
    # PL/Tcl function body
' LANGUAGE 'pltcl';
```

PL/TclU is the same, except that the language should 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' WITH (isStrict);
```

Note the clause `WITH (isStrict)`, which saves us from having to think about NULL input values: if a NULL is passed, the function will not be called at all, but will just return a NULL result automatically.

In a non-strict 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 non-null argument to return the non-null 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 procedure 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 that defines the overpaid_2 function (as found in the older PostgreSQL documentation) in PL/Tcl:

```
CREATE FUNCTION overpaid_2 (EMP) RETURNS bool AS '
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
' LANGUAGE 'pltcl';
```

There is not currently any support for returning a composite-type result value.

24.2.2. Data Values in PL/Tcl

The argument values supplied to a PL/Tcl function's script 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, the PL/Tcl programmer can manipulate data values as if they were just text.

24.2.3. Global Data in PL/Tcl

Sometimes it is useful to have some global status data that is held between two calls to a procedure or is shared between different procedures. This is easily done since all PL/Tcl procedures executed in one backend share the same safe Tcl interpreter. So, any global Tcl variable is accessible to all PL/Tcl procedure calls, and will persist for the duration of the SQL client connection. (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 procedures from unintentionally interfering with each other, a global array is made available to each procedure via the `upvar` command. The global name of this variable is the procedure's internal name and the local name is `GD`. It is recommended that `GD` be used for private status data of a procedure. Use regular Tcl global variables only for values that you specifically intend to be shared among multiple procedures.

An example of using `GD` appears in the `spi_execp` example below.

24.2.4. Database Access from PL/Tcl

The following commands are available to access the database from the body of a PL/Tcl procedure:

`spi_exec ?-count` *`n`*`? ?-array` *`name`*`? ` *`query`* `?`*`loop-body`*`?`

Execute an SQL query given as a string. An error in the query causes an error to be raised. Otherwise, the command's return value is the number of rows processed (selected, inserted, updated, or deleted) by the query, or zero if the query is a utility statement. In addition, if the query 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 query. The effect of this is comparable to setting up the query as a cursor and then saying `FETCH n`.

If the query is a SELECT statement, the values of the SELECT's 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 SELECT column names used as array indexes.

If the query 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 store occurs if the SELECT 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 SELECT result (note: *`loop-body`* is ignored if the given query is not a SELECT). The values of the current row's fields 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 DEBUG 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 field of a SELECT result is NULL, the target variable for it is "unset" rather than being set.

`spi_prepare` *`query typelist`*

Prepares and saves a query plan for later execution. The saved plan will be retained for the life of the current backend.

The query may use *arguments*, which are placeholders for values to be supplied whenever the plan is actually executed. In the query string, refer to arguments by the symbols `$1` ... `$n`. If the query uses arguments, the names of the argument types must be given as a Tcl list. (Write an empty list for *`typelist`* if no arguments are used.) Presently, the argument types must be identified by the internal type names shown in 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` *`name`*? ?`-nulls` *`string`*? *`queryid`* ?*`value-list`*? ?*`loop-body`*?

Execute a query previously prepared with `spi_prepare`. *`queryid`* is the ID returned by `spi_prepare`. If the query references arguments, a *`value-list`* must be supplied: this is a Tcl list of actual values for the arguments. This must be the same length as the argument type list previously given to `spi_prepare`. Omit *`value-list`* if the query has no arguments.

The optional value for `-nulls` is a string of spaces and `'n'` characters telling `spi_execp` which of the arguments are NULLs. If given, it must have exactly the same length as the *`value-list`*. If it is not given, all the argument values are non-NULL.

Except for the way in which the query and its arguments 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 t1_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 t1 WHERE num >= \\$1
 AND num <= \\$2" \\
                [ list int4 int4 ] ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
' LANGUAGE 'pltcl';
```

Note that each backslash that Tcl should see must be doubled when we type in the function, since the main parser processes backslashes too in CREATE FUNCTION. 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`'d or `spi_execp`'d query, if that query was a single-row INSERT. (If not, you get zero.)

`quote` *`string`*

Duplicates 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 queries given to `spi_exec` or `spi_prepare`. For example, think about a query string like

```
"SELECT '$val' AS ret"
```

where the Tcl variable val actually contains `doesn't`. This would result in the final query string

```
SELECT 'doesn't' AS ret
```

which would cause a parse error during `spi_exec` or `spi_prepare`. The submitted query should contain

```
SELECT 'doesn''t' AS ret
```

which can be formed in PL/Tcl as

```
"SELECT '[ quote $val ]' AS ret"
```

One advantage of `spi_execp` is that you don't have to quote argument values like this, since the arguments are never parsed as part of an SQL query string.

`elog` *`level msg`*

Emit a log or error message. Possible levels are `DEBUG`, `NOTICE`, `ERROR`, and `FATAL`. `DEBUG` and `NOTICE` simply emit the given message into the postmaster log (and send it to the client too, in the case of `NOTICE`). `ERROR` raises an error condition: further execution of the function is abandoned, and the current transaction is aborted. `FATAL` aborts the transaction and causes the current backend to shut down (there is probably no good reason to use this error level in PL/Tcl functions, but it's provided for completeness).

24.2.5. Trigger Procedures in PL/Tcl

Trigger procedures can be written in PL/Tcl. As is customary in PostgreSQL, a procedure that's to be called as a trigger must be declared as a function with no arguments and a return type of `opaque`.

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 field names, prefixed with an empty list element. So looking up an element 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 fields are customarily numbered in PostgreSQL.

`$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/UPDATE actions, or empty for DELETE. The array is indexed by field name. Fields that are NULL will not appear in the array!

$OLD

An associative array containing the values of the old table row for UPDATE/DELETE actions, or empty for INSERT. The array is indexed by field name. Fields 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/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 OPAQUE 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 re-used with different tables.

24.2.6. Modules and the `unknown` command

PL/Tcl has support for auto-loading 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 `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.

24.2.7. 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 procedure's pg_proc row 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.

Chapter 25. PL/Perl - Perl Procedural Language

PL/Perl is a loadable procedural language that enables the Perl[1] programming language to be used to write PostgreSQL functions.

25.1. Overview

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 backend or to gain OS-level access under the permissions of the PostgreSQL user ID, as a C function can do. Thus, any unprivileged database user may be permitted to use this language.

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 named `plperlu`). In this case the full Perl language is available. 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.

25.2. Building and Installing PL/Perl

If the `--with-perl` option was supplied to the `configure` script, the PostgreSQL build process will attempt to build the PL/Perl shared library and install it in the PostgreSQL library directory.

On most platforms, since PL/Perl is a shared library, the `libperl` library must be a shared library also. At the time of this writing, this is almost never the case in prebuilt Perl packages. If this difficulty arises in your situation, a message like this will appear during the build to point out this fact:

```
*** 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.
```

If you see this, you will have to re-build and install Perl manually to be able to build PL/Perl. During the configuration process for Perl, request a shared library.

After having reinstalled Perl, change to the directory `src/pl/plperl` in the PostgreSQL source tree and issue the commands

```
gmake clean
gmake all
gmake install
```

to complete the build and installation of the PL/Perl shared library.

To install PL/Perl and/or PL/PerlU in a particular database, use the `createlang` script, for example `createlang plperl` *dbname* or `createlang plperlu` *dbname*.

[1] http://www.perl.com

Tip

If a language is installed into `template1`, all subsequently created databases will have the language installed automatically.

25.3. Description

25.3.1. PL/Perl Functions and Arguments

To create a function in the PL/Perl language, use the standard syntax

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS '
    # PL/Perl function body
' LANGUAGE plperl;
```

PL/PerlU is the same, except that the language should be specified as `plperlu`.

The body of the function is ordinary Perl code. 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 a NULL 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 `WITH (isStrict)` to the function definition to make PostgreSQL do something more reasonable: if a NULL 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 non-null argument to return the non-null argument, rather than NULL:

```
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 a NULL 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;
```

There is not currently any support for returning a composite-type result value.

Tip

Because the function body is passed as an SQL string literal to `CREATE FUNCTION`, you have to escape single quotes and backslashes within your Perl source, typically by doubling them as shown in the above example. Another possible approach is to avoid writing single quotes by using Perl's extended quoting functions (`q[]`, `qq[]`, `qw[]`).

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.

Note that if the same function was created by a superuser using language `plperlu`, execution would succeed.

25.3.2. Data Values in PL/Perl

The argument values supplied to a PL/Perl function's script 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, the PL/Perl programmer can manipulate data values as if they were just text.

25.3.3. Database Access from PL/Perl

Access to the database itself from your Perl function can be done via an experimental module `DBD::PgSPI`[2] (also available at CPAN mirror sites[3]). 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 only one additional Perl command:

[2] http://www.cpan.org/modules/by-module/DBD/APILOS/
[3] http://www.cpan.org/SITES.html

`elog` *`level`*, *`msg`*

Emit a log or error message. Possible levels are `DEBUG`, `NOTICE`, and `ERROR`. `DEBUG` and `NOTICE` simply emit the given message into the postmaster log (and send it to the client too, in the case of `NOTICE`). `ERROR` raises an error condition: further execution of the function is abandoned, and the current transaction is aborted.

25.3.4. Missing Features

PL/Perl functions cannot call each other directly (because they are anonymous subroutines inside Perl). There's presently no way for them to share global variables, either.

PL/Perl cannot currently be used to write trigger functions.

DBD::PgSPI or similar capability should be integrated into the standard PostgreSQL distribution.

Chapter 26. PL/Python - Python Procedural Language

26.1. Introduction

The PL/Python procedural language allows PostgreSQL functions to be written in the Python[1] language.

The current version of PL/Python functions as a trusted language only; access to the file system and other local resources is disabled. Specifically, PL/Python uses the Python restricted execution environment, further restricts it to prevent the use of the file `open` call, and allows only modules from a specific list to be imported. Presently, that list includes: array, bisect, binascii, calendar, cmath, codecs, errno, marshal, math, md5, mpz, operator, pcre, pickle, random, re, regex, sre, sha, string, StringIO, struct, time, whrandom, and zlib.

In the current version, any database error encountered while running a PL/Python function will result in the immediate termination of that function by the server. It is not possible to trap error conditions using Python `try ... catch` constructs. For example, a syntax error in an SQL statement passed to the `plpy.execute()` call will terminate the function. This behavior may be changed in a future release.

26.2. Installation

To build PL/Python, the `--with-python` option needs to be specified when running `configure`. If after building and installing you have a file called `plpython.so` (possibly a different extension), then everything went well. Otherwise you should have seen a notice like this flying by:

```
*** 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.
```

That means you have to rebuild (part of) your Python installation to supply this shared library.

The catch is that the Python distribution or the Python maintainers do not provide any direct way to do this. The closest thing we can offer you is the information in Python FAQ 3.30[2]. On some operating systems you don't really have to build a shared library, but then you will have to convince the PostgreSQL build system of this. Consult the `Makefile` in the `src/pl/plpython` directory for details.

26.3. Using PL/Python

There are sample functions in `plpython_function.sql`. The Python code you write gets transformed into a function. E.g.,

```
CREATE FUNCTION myfunc(text) RETURNS text AS
'return args[0]'
LANGUAGE 'plpython';
```

gets transformed into

```
def __plpython_procedure_myfunc_23456():
```

[1] http://www.python.org
[2] http://www.python.org/doc/FAQ.html#3.30

```
 return args[0]
```

where 23456 is the Oid of the function.

If you do not provide a return value, Python returns the default `None` which may or may not be what you want. The language module translates Python's None into SQL NULL.

PostgreSQL function variables 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` variable and `args[1]` the `integer` variable.

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 backend. Use with care.

Each function gets its own restricted execution object 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.

When a function is used in a trigger, the dictionary TD contains transaction related values. The trigger tuples 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`, or `UNKNOWN`). `TD["level"]` contains one of `ROW`, `STATEMENT`, or `UNKNOWN`. `TD["name"]` contains the trigger name, and `TD["relid"]` contains the relation id of the table on which the trigger occurred. If the trigger was called with arguments they are available in `TD["args"][0]` to `TD["args"][(n -1)]`.

If the trigger “when” is `BEFORE`, you may return `None` or `"OK"` from the Python function to indicate the tuple is unmodified, `"SKIP"` to abort the event, or `"MODIFIED"` to indicate you've modified the tuple.

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.error("msg")`, `plpy.fatal("msg")`, `plpy.debug("msg")`, and `plpy.notice("msg")`. They are mostly equivalent to calling `elog(`*`LEVEL`*`, "msg")`, where *`LEVEL`* is DEBUG, ERROR, FATAL or NOTICE. `plpy.error` and `plpy.fatal` actually raise a Python exception which, if uncaught, causes the PL/Python module to call `elog(ERROR, msg)` when the function handler returns from the Python interpreter. Long jumping out of the Python interpreter is probably not good. `raise plpy.ERROR("msg")` and `raise plpy.FATAL("msg")` are equivalent to calling `plpy.error` or `plpy.fatal`.

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 returned in a result object. The result object emulates a list or dictionary object. The result object can be accessed by row number, and field name. It has these additional methods: `nrows()` which returns the number of rows returned by the query, and `status` which is the `SPI_exec` return variable. The result object can be modified.

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

returns up to 5 rows from my_table. Ff my_table has a column my_field it would be accessed as

```
foo = rv[i]["my_field"]
```

The second function `plpy.prepare` is called with a query string, and a list of argument types if you have bind variables in the query.

```
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 as $1. After preparing you use the function `plpy.execute` to run it.

```
rv = plpy.execute(plan, [ "name" ], 5)
```

The limit argument is optional in the call to `plpy.execute`.

When you prepare a plan using the PL/Python module it is automatically saved. Read the SPI documentation (Chapter 21, *Server Programming Interface*) for a description of what this means. The take home message is if you do

```
plan = plpy.prepare("SOME QUERY")
plan = plpy.prepare("SOME OTHER QUERY")
```

you are leaking memory, as I know of no way to free a saved plan. The alternative of using unsaved plans it even more painful (for me).

PostgreSQL 7.2.8 Reference Manual

The PostgreSQL Global Development Group

PostgreSQL 7.2.8 Reference Manual

The PostgreSQL Global Development Group
Copyright © 1996-2001 The PostgreSQL Global Development Group

Legal Notice

PostgreSQL is Copyright © 1996-2001 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.

Table of Contents

Preface cccxlv
I. SQL Commands 346
ABORT 348
ALTER GROUP 349
ALTER TABLE 350
ALTER USER 354
ANALYZE 356
BEGIN 358
CHECKPOINT 360
CLOSE 361
CLUSTER 362
COMMENT 364
COMMIT 366
COPY 367
CREATE AGGREGATE 373
CREATE CONSTRAINT TRIGGER 376
CREATE DATABASE 377
CREATE FUNCTION 380
CREATE GROUP 384
CREATE INDEX 386
CREATE LANGUAGE 390
CREATE OPERATOR 393
CREATE RULE 397
CREATE SEQUENCE 400
CREATE TABLE 403
CREATE TABLE AS 411
CREATE TRIGGER 413
CREATE TYPE 416
CREATE USER 420
CREATE VIEW 423
DECLARE 425
DELETE 428
DROP AGGREGATE 430
DROP DATABASE 431
DROP FUNCTION 432
DROP GROUP 434
DROP INDEX 435
DROP LANGUAGE 436
DROP OPERATOR 437
DROP RULE 439
DROP SEQUENCE 440
DROP TABLE 441
DROP TRIGGER 443
DROP TYPE 445
DROP USER 447
DROP VIEW 448
END 450
EXPLAIN 451
FETCH 454
GRANT 457
INSERT 460

LISTEN 462
LOAD 464
LOCK 465
MOVE 470
NOTIFY 471
REINDEX 473
RESET 475
REVOKE 476
ROLLBACK 478
SELECT 479
SELECT INTO 490
SET 492
SET CONSTRAINTS 496
SET SESSION AUTHORIZATION 497
SET TRANSACTION 498
SHOW 500
TRUNCATE 501
UNLISTEN 502
UPDATE 504
VACUUM 506
II. PostgreSQL Client Applications 509
createdb 510
createlang 512
createuser 514
dropdb 517
droplang 519
dropuser 521
ecpg 523
pgaccess 527
pg_config 529
pg_dump 531
pg_dumpall 537
pg_restore 539
psql 545
pgtclsh 566
pgtksh 567
vacuumdb 568
III. PostgreSQL Server Applications 571
initdb 572
initlocation 574
ipcclean 575
pg_ctl 576
pg_passwd 579
postgres 580
postmaster 583

Preface

The entries in this *Reference Manual* are meant to provide in reasonable length an authorative, complete, and formal summary about the respective subjects. More information about the use of PostgreSQL, in narrative, tutorial, or example form, may be found in other parts of the PostgreSQL documentation set. See the cross-references listed on each reference page.

The *Reference Manual* entries are also available as traditional “man” pages.

SQL Commands

This part contains reference information for the SQL commands supported by PostgreSQL. By “SQL” the language in general is meant; information about the standards conformance and compatibility of each command can be found on the respective reference page.

Table of Contents

ABORT 348
ALTER GROUP 349
ALTER TABLE 350
ALTER USER 354
ANALYZE 356
BEGIN 358
CHECKPOINT 360
CLOSE 361
CLUSTER 362
COMMENT 364
COMMIT 366
COPY 367
CREATE AGGREGATE 373
CREATE CONSTRAINT TRIGGER 376
CREATE DATABASE 377
CREATE FUNCTION 380
CREATE GROUP 384
CREATE INDEX 386
CREATE LANGUAGE 390
CREATE OPERATOR 393
CREATE RULE 397
CREATE SEQUENCE 400
CREATE TABLE 403
CREATE TABLE AS 411
CREATE TRIGGER 413
CREATE TYPE 416
CREATE USER 420
CREATE VIEW 423
DECLARE 425
DELETE 428
DROP AGGREGATE 430
DROP DATABASE 431
DROP FUNCTION 432
DROP GROUP 434
DROP INDEX 435
DROP LANGUAGE 436
DROP OPERATOR 437
DROP RULE 439
DROP SEQUENCE 440
DROP TABLE 441
DROP TRIGGER 443
DROP TYPE 445
DROP USER 447

DROP VIEW 448
END 450
EXPLAIN 451
FETCH 454
GRANT 457
INSERT 460
LISTEN 462
LOAD 464
LOCK 465
MOVE 470
NOTIFY 471
REINDEX 473
RESET 475
REVOKE 476
ROLLBACK 478
SELECT 479
SELECT INTO 490
SET 492
SET CONSTRAINTS 496
SET SESSION AUTHORIZATION 497
SET TRANSACTION 498
SHOW 500
TRUNCATE 501
UNLISTEN 502
UPDATE 504
VACUUM 506

Name

ABORT — abort the current transaction

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
ABORT [ WORK | TRANSACTION ]
```

Inputs

None.

Outputs

`ROLLBACK`

Message returned if successful.

`NOTICE: ROLLBACK: no transaction in progress`

If there is not any transaction currently in progress.

Description

`ABORT` rolls back the current transaction and causes all the updates made by the transaction to be discarded. This command is identical in behavior to the SQL92 command `ROLLBACK`, and is present only for historical reasons.

Notes

Use `COMMIT` to successfully terminate a transaction.

Usage

To abort all changes:

```
ABORT WORK;
```

Compatibility

SQL92

This command is a PostgreSQL extension present for historical reasons. `ROLLBACK` is the SQL92 equivalent command.

Name

ALTER GROUP — add users to a group or remove users from a group

Synopsis

<refsynopsisdivinfo>2000-01-14</refsynopsisdivinfo>

```
ALTER GROUP name ADD USER username [, ... ]
ALTER GROUP name DROP USER username [, ... ]
```

Inputs

name

The name of the group to modify.

username

Users which are to be added or removed from the group. The user names must exist.

Outputs

`ALTER GROUP`

Message returned if the alteration was successful.

Description

`ALTER GROUP` is used to add or remove users from a group. Only database superusers can use this command. Adding a user to a group does not create the user. Similarly, removing a user from a group does not drop the user itself.

Use CREATE GROUP to create a new group and DROP GROUP to remove a group.

Usage

Add users to a group:

```
ALTER GROUP staff ADD USER karl, john;
```

Remove a user from a group:

```
ALTER GROUP workers DROP USER beth;
```

Compatibility

SQL92

There is no `ALTER GROUP` statement in SQL92. The concept of roles is similar.

Name

ALTER TABLE — change the definition of a table

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
ALTER TABLE [ ONLY ] table [ * ]
    ADD [ COLUMN ] column type [ column_constraint [ ... ] ]
ALTER TABLE [ ONLY ] table [ * ]
    ALTER [ COLUMN ] column { SET DEFAULT value | DROP DEFAULT }
ALTER TABLE [ ONLY ] table [ * ]
    ALTER [ COLUMN ] column SET STATISTICS integer
ALTER TABLE [ ONLY ] table [ * ]
    RENAME [ COLUMN ] column TO newcolumn
ALTER TABLE table
    RENAME TO new_table
ALTER TABLE table
    ADD table_constraint_definition
ALTER TABLE [ ONLY ] table
 DROP CONSTRAINT constraint { RESTRICT | CASCADE }
ALTER TABLE table
 OWNER TO new_owner
```

Inputs

table

The name of an existing table to alter.

column

Name of a new or existing column.

type

Type of the new column.

newcolumn

New name for an existing column.

new_table

New name for the table.

table_constraint_definition

New table constraint for the table

new_owner

The user name of the new owner of the table.

Outputs

`ALTER`

Message returned from column or table renaming.

`ERROR`

Message returned if table or column is not available.

Description

`ALTER TABLE` changes the definition of an existing table. The `ADD COLUMN` form adds a new column to the table using the same syntax as CREATE TABLE. The `ALTER COLUMN SET/DROP DEFAULT` forms allow you to set or remove the default for the column. Note that defaults only apply to subsequent `INSERT` commands; they do not cause rows already in the table to change. The `ALTER COLUMN SET STATISTICS` form allows you to set the statistics-gathering target for subsequent ANALYZE operations. The `RENAME` clause causes the name of a table, column, index, or sequence to change without changing any of the data. The data will remain of the same type and size after the command is executed. The ADD *`table_constraint_definition`* clause adds a new constraint to the table using the same syntax as CREATE TABLE. The DROP CONSTRAINT *`constraint`* clause drops all constraints on the table (and its children) that match *`constraint`*. The OWNER clause changes the owner of the table to the user *`new user`*.

You must own the table in order to change its schema.

Notes

The keyword `COLUMN` is noise and can be omitted.

In the current implementation of `ADD COLUMN`, default and NOT NULL clauses for the new column are not supported. You can use the `SET DEFAULT` form of `ALTER TABLE` to set the default later. (You may also want to update the already existing rows to the new default value, using UPDATE.)

In DROP CONSTRAINT, the RESTRICT keyword is required, although dependencies are not yet checked. The CASCADE option is unsupported. Currently DROP CONSTRAINT drops only CHECK constraints. To remove a PRIMARY or UNIQUE constraint, drop the relevant index using the DROP INDEX command. To remove FOREIGN KEY constraints you need to recreate and reload the table, using other parameters to the CREATE TABLE command.

For example, to drop all constraints on a table `distributors`:

```
CREATE TABLE temp AS SELECT * FROM distributors;
DROP TABLE distributors;
CREATE TABLE distributors AS SELECT * FROM temp;
DROP TABLE temp;
```

You must own the table in order to change it. Changing any part of the schema of a system catalog is not permitted. The *PostgreSQL User's Guide* has further information on inheritance.

Refer to `CREATE TABLE` for a further description of valid arguments.

Usage

To add a column of type `varchar` to a table:

```
ALTER TABLE distributors ADD COLUMN address VARCHAR(30);
```

To rename an existing column:

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

To rename an existing table:

```
ALTER TABLE distributors RENAME TO suppliers;
```

To add a check constraint to a table:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK
 (char_length(zipcode) = 5);
```

To remove a check constraint from a table and all its children:

```
ALTER TABLE distributors DROP CONSTRAINT zipchk RESTRICT;
```

To add a foreign key constraint to a table:

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address)
 REFERENCES addresses(address) MATCH FULL;
```

To add a (multicolumn) unique constraint to a table:

```
ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE
 (dist_id, zipcode);
```

To add an automatically named primary key constraint to a table, noting that a table can only ever have one primary key:

```
ALTER TABLE distributors ADD PRIMARY KEY (dist_id);
```

Compatibility

SQL92

The `ADD COLUMN` form is compliant with the exception that it does not support defaults and NOT NULL constraints, as explained above. The `ALTER COLUMN` form is in full compliance.

SQL92 specifies some additional capabilities for `ALTER TABLE` statement which are not yet directly supported by PostgreSQL:

```
ALTER TABLE table DROP [ COLUMN ] column { RESTRICT |
 CASCADE }
```

Removes a column from a table. Currently, to remove an existing column the table must be recreated and reloaded:

```
CREATE TABLE temp AS SELECT did, city FROM distributors;
DROP TABLE distributors;
CREATE TABLE distributors (
    did      DECIMAL(3)  DEFAULT 1,
    name     VARCHAR(40) NOT NULL
);
INSERT INTO distributors SELECT * FROM temp;
DROP TABLE temp;
```

The clauses to rename tables, columns, indexes, and sequences are PostgreSQL extensions from SQL92.

Name

ALTER USER — change a database user account

Synopsis

<refsynopsisdivinfo>2001-07-10</refsynopsisdivinfo>

```
ALTER USER username [ [ WITH ] option [ ... ] ]

where option can be:

    [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
         | CREATEDB | NOCREATEDB
         | CREATEUSER | NOCREATEUSER
         | VALID UNTIL 'abstime'
```

Inputs

username

The name of the user whose details are to be altered.

password

The new password to be used for this account.

ENCRYPTED
UNENCRYPTED

These keywords control whether the password is stored encrypted in `pg_shadow`. (See CREATE USER for more information about this choice.)

CREATEDB
NOCREATEDB

These clauses define a user's ability to create databases. If CREATEDB is specified, the user being defined will be allowed to create his own databases. Using NOCREATEDB will deny a user the ability to create databases.

CREATEUSER
NOCREATEUSER

These clauses determine whether a user will be permitted to create new users himself. This option will also make the user a superuser who can override all access restrictions.

abstime

The date (and, optionally, the time) at which this user's password is to expire.

Outputs

`ALTER USER`

Message returned if the alteration was successful.

```
ERROR: ALTER USER: user "username" does not exist
```

Error message returned if the specified user is not known to the database.

Description

`ALTER USER` is used to change the attributes of a user's PostgreSQL account. Attributes not mentioned in the command retain their previous settings.

Only a database superuser can change privileges and password expiration with this command. Ordinary users can only change their own password.

`ALTER USER` cannot change a user's group memberships. Use ALTER GROUP to do that.

Use CREATE USER to create a new user and DROP USER to remove a user.

Usage

Change a user password:

```
ALTER USER davide WITH PASSWORD 'hu8jmn3';
```

Change a user's valid until date:

```
ALTER USER manuel VALID UNTIL 'Jan 31 2030';
```

Change a user's valid until date, specifying that his authorization should expire at midday on 4th May 1998 using the time zone which is one hour ahead of UTC:

```
ALTER USER chris VALID UNTIL 'May 4 12:00:00 1998 +1';
```

Give a user the ability to create other users and new databases:

```
ALTER USER miriam CREATEUSER CREATEDB;
```

Compatibility

SQL92

There is no `ALTER USER` statement in SQL92. The standard leaves the definition of users to the implementation.

Name

ANALYZE — collect statistics about a database

Synopsis

<refsynopsisdivinfo>2001-05-04</refsynopsisdivinfo>

```
ANALYZE [ VERBOSE ] [ table [ (column [, ...] ) ] ]
```

Inputs

VERBOSE

Enables display of progress messages.

table

The name of a specific table to analyze. Defaults to all tables.

column

The name of a specific column to analyze. Defaults to all columns.

Outputs

ANALYZE

The command is complete.

Description

ANALYZE collects statistics about the contents of PostgreSQL tables, and stores the results in the system table pg_statistic. Subsequently, the query planner uses the statistics to help determine the most efficient execution plans for queries.

With no parameter, ANALYZE examines every table in the current database. With a parameter, ANALYZE examines only that table. It is further possible to give a list of column names, in which case only the statistics for those columns are updated.

Notes

It is a good idea to run ANALYZE periodically, or just after making major changes in the contents of a table. Accurate statistics will help the planner to choose the most appropriate query plan, and thereby improve the speed of query processing. A common strategy is to run VACUUM and ANALYZE once a day during a low-usage time of day.

Unlike VACUUM FULL, ANALYZE requires only a read lock on the target table, so it can run in parallel with other activity on the table.

For large tables, ANALYZE takes a random sample of the table contents, rather than examining every row. This allows even very large tables to be analyzed in a small amount of time. Note however that the statistics are only approximate, and will change slightly each time ANALYZE is run, even if the actual table contents did not change. This may result in small changes in the planner's estimated costs shown by EXPLAIN.

The collected statistics usually include a list of some of the most common values in each column and a histogram showing the approximate data distribution in each column. One or both of these may be omitted if `ANALYZE` deems them uninteresting (for example, in a unique-key column, there are no common values) or if the column data type does not support the appropriate operators. There is more information about the statistics in the *User's Guide*.

The extent of analysis can be controlled by adjusting the per-column statistics target with `ALTER TABLE ALTER COLUMN SET STATISTICS` (see ALTER TABLE). The target value sets the maximum number of entries in the most-common-value list and the maximum number of bins in the histogram. The default target value is 10, but this can be adjusted up or down to trade off accuracy of planner estimates against the time taken for `ANALYZE` and the amount of space occupied in `pg_statistic`. In particular, setting the statistics target to zero disables collection of statistics for that column. It may be useful to do that for columns that are never used as part of the WHERE, GROUP BY, or ORDER BY clauses of queries, since the planner will have no use for statistics on such columns.

The largest statistics target among the columns being analyzed determines the number of table rows sampled to prepare the statistics. Increasing the target causes a proportional increase in the time and space needed to do `ANALYZE`.

Compatibility

SQL92

There is no `ANALYZE` statement in SQL92.

Name

BEGIN — start a transaction block

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
BEGIN [ WORK | TRANSACTION ]
```

Inputs

WORK
TRANSACTION

Optional keywords. They have no effect.

Outputs

`BEGIN`

This signifies that a new transaction has been started.

`NOTICE: BEGIN: already a transaction in progress`

This indicates that a transaction was already in progress. The current transaction is not affected.

Description

By default, PostgreSQL executes transactions in *unchained mode* (also known as "autocommit" in other database systems). In other words, each user statement is executed in its own transaction and a commit is implicitly performed at the end of the statement (if execution was successful, otherwise a rollback is done). `BEGIN` initiates a user transaction in chained mode, i.e., all user statements after `BEGIN` command will be executed in a single transaction until an explicit COMMIT , ROLLBACK , or execution abort. Statements in chained mode are executed much faster, because transaction start/commit requires significant CPU and disk activity. Execution of multiple statements inside a transaction is also required for consistency when changing several related tables.

The default transaction isolation level in PostgreSQL is READ COMMITTED, where queries inside the transaction see only changes committed before query execution. So, you have to use `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE` just after `BEGIN` if you need more rigorous transaction isolation. In SERIALIZABLE mode queries will see only changes committed before the entire transaction began (actually, before execution of the first DML statement in a serializable transaction).

If the transaction is committed, PostgreSQL will ensure either that all updates are done or else that none of them are done. Transactions have the standard ACID (atomic, consistent, isolatable, and durable) property.

Notes

Refer to LOCK for further information about locking tables inside a transaction.

Use COMMIT or ROLLBACK to terminate a transaction.

Usage

To begin a user transaction:

```
BEGIN WORK;
```

Compatibility

SQL92

`BEGIN` is a PostgreSQL language extension. There is no explicit `BEGIN` command in SQL92; transaction initiation is always implicit and it terminates either with a `COMMIT` or `ROLLBACK` statement.

Note

Many relational database systems offer an autocommit feature as a convenience.

Incidentally, the `BEGIN` keyword is used for a different purpose in embedded SQL. You are advised to be careful about the transaction semantics when porting database applications.

SQL92 also requires SERIALIZABLE to be the default transaction isolation level.

<docinfo>2001-01-24</docinfo>

Name

CHECKPOINT — force a transaction log checkpoint

Synopsis

```
CHECKPOINT
```

Description

Write-Ahead Logging (WAL) puts a checkpoint in the transaction log every so often. (To adjust the automatic checkpoint interval, see the run-time configuration options *CHECKPOINT_SEGMENTS* and *CHECKPOINT_TIMEOUT*.) The CHECKPOINT command forces an immediate checkpoint when the command is issued, without waiting for a scheduled checkpoint.

A checkpoint is a point in the transaction log sequence at which all data files have been updated to reflect the information in the log. All data files will be flushed to disk. Refer to the *PostgreSQL Administrator's Guide* for more information about the WAL system.

Only superusers may call CHECKPOINT. The command is not intended for use during normal operation.

See Also

PostgreSQL Administrator's Guide

Compatibility

The CHECKPOINT command is a PostgreSQL language extension.

Name

CLOSE — close a cursor

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
CLOSE cursor
```

Inputs

cursor

The name of an open cursor to close.

Outputs

`CLOSE`

Message returned if the cursor is successfully closed.

`NOTICE PerformPortalClose: portal "`*cursor*`" not found`

This warning is given if *cursor* is not declared or has already been closed.

Description

`CLOSE` frees the resources associated with an open cursor. After the cursor is closed, no subsequent operations are allowed on it. A cursor should be closed when it is no longer needed.

An implicit close is executed for every open cursor when a transaction is terminated by `COMMIT` or `ROLLBACK`.

Notes

PostgreSQL does not have an explicit `OPEN` cursor statement; a cursor is considered open when it is declared. Use the `DECLARE` statement to declare a cursor.

Usage

Close the cursor `liahona`:

```
CLOSE liahona;
```

Compatibility

SQL92

`CLOSE` is fully compatible with SQL92.

Name

CLUSTER — cluster a table according to an index

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
CLUSTER indexname ON tablename
```

Inputs

indexname

The name of an index.

table

The name of a table.

Outputs

CLUSTER

The clustering was done successfully.

ERROR: relation <*tablerelation_number*> inherits "*table*"

ERROR: Relation *table* does not exist!

Description

CLUSTER instructs PostgreSQL to cluster the table specified by *table* approximately based on the index specified by *indexname*. The index must already have been defined on *tablename*.

When a table is clustered, it is physically reordered based on the index information. The clustering is static. In other words, as the table is updated, the changes are not clustered. No attempt is made to keep new instances or updated tuples clustered. If one wishes, one can re-cluster manually by issuing the command again.

Notes

The table is actually copied to a temporary table in index order, then renamed back to the original name. For this reason, all grant permissions and other indexes are lost when clustering is performed.

In cases where you are accessing single rows randomly within a table, the actual order of the data in the heap table is unimportant. However, if you tend to access some data more than others, and there is an index that groups them together, you will benefit from using CLUSTER.

Another place where CLUSTER is helpful is in cases where you use an index to pull out several rows from a table. If you are requesting a range of indexed values from a table, or a single indexed value that has multiple rows that match, CLUSTER will help because once the index identifies the heap page for the

first row that matches, all other rows that match are probably already on the same heap page, saving disk accesses and speeding up the query.

There are two ways to cluster data. The first is with the `CLUSTER` command, which reorders the original table with the ordering of the index you specify. This can be slow on large tables because the rows are fetched from the heap in index order, and if the heap table is unordered, the entries are on random pages, so there is one disk page retrieved for every row moved. PostgreSQL has a cache, but the majority of a big table will not fit in the cache.

Another way to cluster data is to use

```
SELECT columnlist INTO TABLE newtable
     FROM table ORDER BY columnlist
```

which uses the PostgreSQL sorting code in the ORDER BY clause to match the index, and which is much faster for unordered data. You then drop the old table, use `ALTER TABLE...RENAME` to rename *`newtable`* to the old name, and recreate the table's indexes. The only problem is that OIDs will not be preserved. From then on, `CLUSTER` should be fast because most of the heap data has already been ordered, and the existing index is used.

Usage

Cluster the employees relation on the basis of its salary attribute:

```
CLUSTER emp_ind ON emp;
```

Compatibility

SQL92

There is no `CLUSTER` statement in SQL92.

Name

COMMENT — define or change the comment of an object

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
COMMENT ON
[
  [ DATABASE | INDEX | RULE | SEQUENCE | TABLE | TYPE |
 VIEW ] object_name |
  COLUMN table_name.column_name |
  AGGREGATE agg_name (agg_type) |
  FUNCTION func_name (arg1, arg2, ...) |
  OPERATOR op (leftoperand_type rightoperand_type) |
  TRIGGER trigger_name ON table_name
] IS 'text'
```

Inputs

object_name, *table_name*, *column_name*, *agg_name*, *func_name*, *op*, *trigger_name*

The name of the object to be be commented.

text

The comment to add.

Outputs

COMMENT

Message returned if the table is successfully commented.

Description

COMMENT stores a comment about a database object. Comments can be easily retrieved with psql's \dd, \d+, or \l+ commands. Other user interfaces to retrieve comments can be built atop the same built-in functions that psql uses, namely obj_description() and col_description().

To modify a comment, issue a new COMMENT command for the same object. Only one comment string is stored for each object. To remove a comment, write NULL in place of the text string. Comments are automatically dropped when the object is dropped.

It should be noted that there is presently no security mechanism for comments: any user connected to a database can see all the comments for objects in that database (although only superusers can change comments for objects that they don't own). Therefore, don't put security-critical information in comments.

Usage

Comment the table mytable:

```
COMMENT ON mytable IS 'This is my table.';
```

Some more examples:

```
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee id';
COMMENT ON RULE my_rule IS 'Logs UPDATES of employee records';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON TABLE my_table IS 'Employee Information';
COMMENT ON TYPE my_type IS 'Complex Number support';
COMMENT ON VIEW my_view IS 'View of departmental costs';
COMMENT ON COLUMN my_table.my_field IS 'Employee ID number';
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes
 sample variance';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman
 Numeral';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two
 text';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for R.I.';
```

Compatibility

SQL92

There is no `COMMENT` in SQL92.

Name

COMMIT — commit the current transaction

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
COMMIT [ WORK | TRANSACTION ]
```

Inputs

WORK
TRANSACTION

Optional keywords. They have no effect.

Outputs

`COMMIT`

Message returned if the transaction is successfully committed.

`NOTICE: COMMIT: no transaction in progress`

If there is no transaction in progress.

Description

`COMMIT` commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

Notes

The keywords WORK and TRANSACTION are noise and can be omitted.

Use ROLLBACK to abort a transaction.

Usage

To make all changes permanent:

```
COMMIT WORK;
```

Compatibility

SQL92

SQL92 only specifies the two forms `COMMIT` and `COMMIT WORK`. Otherwise full compatibility.

Name

COPY — copy data between files and tables

Synopsis

<refsynopsisdivinfo>1999-12-11</refsynopsisdivinfo>

```
COPY [ BINARY ] table [ WITH OIDS ]
    FROM { 'filename' | stdin }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]
COPY [ BINARY ] table [ WITH OIDS ]
    TO { 'filename' | stdout }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]
```

Inputs

BINARY

Changes the behavior of field formatting, forcing all data to be stored or read in binary format rather than as text. The DELIMITERS and WITH NULL options are irrelevant for binary format.

table

The name of an existing table.

WITH OIDS

Specifies copying the internal object id (OID) for each row.

filename

The absolute Unix path name of the input or output file.

stdin

Specifies that input comes from the client application.

stdout

Specifies that output goes to the client application.

delimiter

The character that separates fields within each row (line) of the file.

null string

The string that represents a NULL value. The default is "\N" (backslash-N). You might prefer an empty string, for example.

Note

On a copy in, any data item that matches this string will be stored as a NULL value, so you should make sure that you use the same string as you used on copy out.

Outputs

`COPY`

The copy completed successfully.

`ERROR:` *reason*

The copy failed for the reason stated in the error message.

Description

`COPY` moves data between PostgreSQL tables and standard file-system files. `COPY TO` copies the entire contents of a table to a file, while `COPY FROM` copies data from a file to a table (appending the data to whatever is in the table already).

`COPY` with a file name instructs the PostgreSQL backend to directly read from or write to a file. The file must be accessible to the backend and the name must be specified from the viewpoint of the backend. When `stdin` or `stdout` is specified, data flows through the client frontend to the backend.

Tip

Do not confuse `COPY` with the psql instruction `\copy`. `\copy` invokes `COPY FROM stdin` or `COPY TO stdout`, and then fetches/stores the data in a file accessible to the psql client. Thus, file accessibility and access rights depend on the client rather than the backend when `\copy` is used.

Notes

`COPY` can only be used with plain tables, not with views.

The BINARY keyword will force all data to be stored/read as binary format rather than as text. It is somewhat faster than the normal copy command, but a binary copy file is not portable across machine architectures.

By default, a text copy uses a tab ("\t") character as a delimiter between fields. The field delimiter may be changed to any other single character with the keyword phrase USING DELIMITERS. Characters in data fields that happen to match the delimiter character will be backslash quoted.

You must have *select access* on any table whose values are read by `COPY`, and either *insert* or *update access* to a table into which values are being inserted by `COPY`. The backend also needs appropriate Unix permissions for any file read or written by `COPY`.

`COPY TO` neither invokes rules nor acts on column defaults. It does invoke triggers and check constraints.

`COPY` stops operation at the first error. This should not lead to problems in the event of a `COPY FROM`, but the target relation will already have received earlier rows in a `COPY TO`. These rows will not be visible or accessible, but they still occupy disk space. This may amount to a considerable amount of wasted disk space if the failure happened well into a large copy operation. You may wish to invoke `VACUUM` to recover the wasted space.

Files named in a `COPY` command are read or written directly by the backend, not by the client application. Therefore, they must reside on or be accessible to the database server machine, not the client. They must be accessible to and readable or writable by the PostgreSQL user (the user ID the server runs as), not the client. `COPY` naming a file is only allowed to database superusers, since it allows reading or writing any file that the backend has privileges to access.

Tip

The psql instruction `\copy` reads or writes files on the client machine with the client's permissions, so it is not restricted to superusers.

It is recommended that the filename used in `COPY` always be specified as an absolute path. This is enforced by the backend in the case of `COPY TO`, but for `COPY FROM` you do have the option of reading from a file specified by a relative path. The path will be interpreted relative to the backend's working directory (somewhere below `$PGDATA`), not the client's working directory.

File Formats

Text Format

When `COPY` is used without the BINARY option, the file read or written is a text file with one line per table row. Columns (attributes) in a row are separated by the delimiter character. The attribute values themselves are strings generated by the output function, or acceptable to the input function, of each attribute's data type. The specified null-value string is used in place of attributes that are NULL.

If WITH OIDS is specified, the OID is read or written as the first column, preceding the user data columns. (An error is raised if WITH OIDS is specified for a table that does not have OIDs.)

End of data can be represented by a single line containing just backslash-period (`\.`). An end-of-data marker is not necessary when reading from a Unix file, since the end of file serves perfectly well; but an end marker must be provided when copying data to or from a client application.

Backslash characters (`\`) may be used in the `COPY` data to quote data characters that might otherwise be taken as row or column delimiters. In particular, the following characters *must* be preceded by a backslash if they appear as part of an attribute value: backslash itself, newline, and the current delimiter character.

The following special backslash sequences are recognized by `COPY FROM`:

Sequence	Represents
`\b`	Backspace (ASCII 8)
`\f`	Form feed (ASCII 12)
`\n`	Newline (ASCII 10)
`\r`	Carriage return (ASCII 13)
`\t`	Tab (ASCII 9)
`\v`	Vertical tab (ASCII 11)
`\`*`digits`*	Backslash followed by one to three octal digits specifies the character with that numeric code

Presently, `COPY TO` will never emit an octal-digits backslash sequence, but it does use the other sequences listed above for those control characters.

Never put a backslash before a data character `N` or period (`.`). Such pairs will be mistaken for the default null string or the end-of-data marker, respectively. Any other backslashed character that is not mentioned in the above table will be taken to represent itself.

It is strongly recommended that applications generating COPY data convert data newlines and carriage returns to the `\n` and `\r` sequences respectively. At present (PostgreSQL 7.2 and older versions) it is

possible to represent a data carriage return without any special quoting, and to represent a data newline by a backslash and newline. However, these representations will not be accepted by default in future releases.

Note that the end of each row is marked by a Unix-style newline ("\n"). Presently, `COPY FROM` will not behave as desired if given a file containing DOS- or Mac-style newlines. This is expected to change in future releases.

Binary Format

The file format used for `COPY BINARY` changed in PostgreSQL v7.1. The new format consists of a file header, zero or more tuples, and a file trailer.

File Header

The file header consists of 24 bytes of fixed fields, followed by a variable-length header extension area. The fixed fields are:

Signature

12-byte sequence `PGBCOPY\n\377\r\n\0` --- note that the null is a required part of the signature. (The signature is designed to allow easy identification of files that have been munged by a non-8-bit-clean transfer. This signature will be changed by newline-translation filters, dropped nulls, dropped high bits, or parity changes.)

Integer layout field

int32 constant 0x01020304 in source's byte order. Potentially, a reader could engage in byte-flipping of subsequent fields if the wrong byte order is detected here.

Flags field

int32 bit mask to denote important aspects of the file format. Bits are numbered from 0 (LSB) to 31 (MSB) --- note that this field is stored with source's endianness, as are all subsequent integer fields. Bits 16-31 are reserved to denote critical file format issues; a reader should abort if it finds an unexpected bit set in this range. Bits 0-15 are reserved to signal backwards-compatible format issues; a reader should simply ignore any unexpected bits set in this range. Currently only one flag bit is defined, and the rest must be zero:

Bit 16

if 1, OIDs are included in the dump; if 0, not

Header extension area length

int32 length in bytes of remainder of header, not including self. In the initial version this will be zero, and the first tuple follows immediately. Future changes to the format might allow additional data to be present in the header. A reader should silently skip over any header extension data it does not know what to do with.

The header extension area is envisioned to contain a sequence of self-identifying chunks. The flags field is not intended to tell readers what is in the extension area. Specific design of header extension contents is left for a later release.

This design allows for both backwards-compatible header additions (add header extension chunks, or set low-order flag bits) and non-backwards-compatible changes (set high-order flag bits to signal such changes, and add supporting data to the extension area if needed).

Tuples

Each tuple begins with an int16 count of the number of fields in the tuple. (Presently, all tuples in a table will have the same count, but that might not always be true.) Then, repeated for each field in the tuple, there is an int16 typlen word possibly followed by field data. The typlen field is interpreted thus:

Zero

Field is NULL. No data follows.

> 0

Field is a fixed-length data type. Exactly N bytes of data follow the typlen word.

-1

Field is a varlena data type. The next four bytes are the varlena header, which contains the total value length including itself.

< -1

Reserved for future use.

For non-NULL fields, the reader can check that the typlen matches the expected typlen for the destination column. This provides a simple but very useful check that the data is as expected.

There is no alignment padding or any other extra data between fields. Note also that the format does not distinguish whether a data type is pass-by-reference or pass-by-value. Both of these provisions are deliberate: they might help improve portability of the files (although of course endianness and floating-point-format issues can still keep you from moving a binary file across machines).

If OIDs are included in the dump, the OID field immediately follows the field-count word. It is a normal field except that it's not included in the field-count. In particular it has a typlen --- this will allow handling of 4-byte vs 8-byte OIDs without too much pain, and will allow OIDs to be shown as NULL if that ever proves desirable.

File Trailer

The file trailer consists of an int16 word containing -1. This is easily distinguished from a tuple's field-count word.

A reader should report an error if a field-count word is neither -1 nor the expected number of columns. This provides an extra check against somehow getting out of sync with the data.

Usage

The following example copies a table to standard output, using a vertical bar (|) as the field delimiter:

```
COPY country TO stdout USING DELIMITERS '|';
```

To copy data from a Unix file into a table country:

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

Here is a sample of data suitable for copying into a table from `stdin` (so it has the termination sequence on the last line):

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
\.
```

Note that the white space on each line is actually a TAB.

The following is the same data, output in binary format on a Linux/i586 machine. The data is shown after filtering through the Unix utility `od -c`. The table has three fields; the first is `char(2)`, the second is `text`, and the third is `integer`. All the rows have a null value in the third field.

```
0000000   P   G   B   C   O   P   Y  \n 377  \r  \n  \0 004 003 002
 001
0000020  \0  \0  \0  \0  \0  \0  \0  \0 003  \0 377 377 006  \0  \0
 \0
0000040   A   F 377 377 017  \0  \0  \0   A   F   G   H   A   N   I
 S
0000060   T   A   N  \0  \0 003  \0 377 377 006  \0  \0  \0   A   L
 377
0000100 377  \v  \0  \0  \0   A   L   B   A   N   I   A  \0  \0 003
 \0
0000120 377 377 006  \0  \0  \0   D   Z 377 377  \v  \0  \0  \0   A
 L
0000140   G   E   R   I   A  \0  \0 003  \0 377 377 006  \0  \0  \0
 Z
0000160   M 377 377  \n  \0  \0  \0   Z   A   M   B   I   A  \0  \0
 003
0000200  \0 377 377 006  \0  \0  \0   Z   W 377 377  \f  \0  \0  \0
 Z
0000220   I   M   B   A   B   W   E  \0  \0 377 377
```

Compatibility

SQL92

There is no `COPY` statement in SQL92.

Name

CREATE AGGREGATE — define a new aggregate function

Synopsis

<refsynopsisdivinfo>2000-07-16</refsynopsisdivinfo>

```
CREATE AGGREGATE name ( BASETYPE = input_data_type,
    SFUNC = sfunc, STYPE = state_type
    [ , FINALFUNC = ffunc ]
    [ , INITCOND = initial_condition ] )
```

Inputs

name

The name of an aggregate function to create.

input_data_type

The input data type on which this aggregate function operates. This can be specified as ANY for an aggregate that does not examine its input values (an example is `count(*)`).

sfunc

The name of the state transition function to be called for each input data value. This is normally a function of two arguments, the first being of type *state_type* and the second of type *input_data_type*. Alternatively, for an aggregate that does not examine its input values, the function takes just one argument of type *state_type*. In either case the function must return a value of type *state_type*. This function takes the current state value and the current input data item, and returns the next state value.

state_type

The data type for the aggregate's state value.

ffunc

The name of the final function called to compute the aggregate's result after all input data has been traversed. The function must take a single argument of type *state_type*. The output data type of the aggregate is defined as the return type of this function. If *ffunc* is not specified, then the ending state value is used as the aggregate's result, and the output type is *state_type*.

initial_condition

The initial setting for the state value. This must be a literal constant in the form accepted for the data type *state_type*. If not specified, the state value starts out NULL.

Outputs

CREATE

Message returned if the command completes successfully.

Description

`CREATE AGGREGATE` allows a user or programmer to extend PostgreSQL functionality by defining new aggregate functions. Some aggregate functions for base types such as `min(integer)` and `avg(double precision)` are already provided in the base distribution. If one defines new types or needs an aggregate function not already provided, then `CREATE AGGREGATE` can be used to provide the desired features.

An aggregate function is identified by its name and input data type. Two aggregates can have the same name if they operate on different input types. To avoid confusion, do not make an ordinary function of the same name and input data type as an aggregate.

An aggregate function is made from one or two ordinary functions: a state transition function *`sfunc`*, and an optional final calculation function *`ffunc`*. These are used as follows:

```
sfunc( internal-state, next-data-item ) ---> next-internal-state
ffunc( internal-state ) ---> aggregate-value
```

PostgreSQL creates a temporary variable of data type *`stype`* to hold the current internal state of the aggregate. At each input data item, the state transition function is invoked to calculate a new internal state value. After all the data has been processed, the final function is invoked once to calculate the aggregate's output value. If there is no final function then the ending state value is returned as-is.

An aggregate function may provide an initial condition, that is, an initial value for the internal state value. This is specified and stored in the database as a field of type `text`, but it must be a valid external representation of a constant of the state value data type. If it is not supplied then the state value starts out NULL.

If the state transition function is declared “strict”, then it cannot be called with NULL inputs. With such a transition function, aggregate execution behaves as follows. NULL input values are ignored (the function is not called and the previous state value is retained). If the initial state value is NULL, then the first non-NULL input value replaces the state value, and the transition function is invoked beginning with the second non-NULL input value. This is handy for implementing aggregates like `max`. Note that this behavior is only available when *`state_type`* is the same as *`input_data_type`*. When these types are different, you must supply a non-NULL initial condition or use a non-strict transition function.

If the state transition function is not strict, then it will be called unconditionally at each input value, and must deal with NULL inputs and NULL transition values for itself. This allows the aggregate author to have full control over the aggregate's handling of NULLs.

If the final function is declared “strict”, then it will not be called when the ending state value is NULL; instead a NULL result will be output automatically. (Of course this is just the normal behavior of strict functions.) In any case the final function has the option of returning NULL. For example, the final function for `avg` returns NULL when it sees there were zero input tuples.

Notes

Use `DROP AGGREGATE` to drop aggregate functions.

The parameters of `CREATE AGGREGATE` can be written in any order, not just the order illustrated above.

Usage

Refer to the chapter on aggregate functions in the *PostgreSQL Programmer's Guide* for complete examples of usage.

Compatibility

SQL92

`CREATE AGGREGATE` is a PostgreSQL language extension. There is no `CREATE AGGREGATE` in SQL92.

Name

CREATE CONSTRAINT TRIGGER — define a new constraint trigger

Synopsis

<refsynopsisdivinfo>2000-04-13</refsynopsisdivinfo>

```
CREATE CONSTRAINT TRIGGER name
    AFTER events ON
    relation constraint attributes
    FOR EACH ROW EXECUTE PROCEDURE func '(' args ')'
```

Inputs

name

The name of the constraint trigger.

events

The event categories for which this trigger should be fired.

relation

Table name of the triggering relation.

constraint

Actual constraint specification.

attributes

Constraint attributes.

func(*args*)

Function to call as part of the trigger processing.

Outputs

CREATE CONSTRAINT

Message returned if successful.

Description

CREATE CONSTRAINT TRIGGER is used from inside of CREATE/ALTER TABLE and by pg_dump to create the special triggers for referential integrity.

It is not intended for general use.

Name

CREATE DATABASE — create a new database

Synopsis

<refsynopsisdivinfo>1999-12-11</refsynopsisdivinfo>

```
CREATE DATABASE name
    [ WITH [ LOCATION = 'dbpath' ]
           [ TEMPLATE = template ]
           [ ENCODING = encoding ] ]
```

Inputs

name

The name of a database to create.

dbpath

An alternate file-system location in which to store the new database, specified as a string literal; or DEFAULT to use the default location.

template

Name of template from which to create the new database, or DEFAULT to use the default template (template1).

encoding

Multibyte encoding method to use in the new database. Specify a string literal name (e.g., 'SQL_ASCII'), or an integer encoding number, or DEFAULT to use the default encoding.

Outputs

CREATE DATABASE

Message returned if the command completes successfully.

ERROR: user '*username*' is not allowed to create/drop databases

You must have the special CREATEDB privilege to create databases. See CREATE USER .

ERROR: createdb: database "*name*" already exists

This occurs if a database with the *name* specified already exists.

ERROR: database path may not contain single quotes

The database location *dbpath* cannot contain single quotes. This is required so that the shell commands that create the database directory can execute safely.

ERROR: CREATE DATABASE: may not be called in a transaction block

If you have an explicit transaction block in progress you cannot call CREATE DATABASE. You must finish the transaction first.

```
ERROR: Unable to create database directory 'path'.
ERROR: Could not initialize database directory.
```

These are most likely related to insufficient permissions on the data directory, a full disk, or other file system problems. The user under which the database server is running must have access to the location.

Description

`CREATE DATABASE` creates a new PostgreSQL database. The creator becomes the owner of the new database.

An alternate location can be specified in order to, for example, store the database on a different disk. The path must have been prepared with the initlocation command.

If the path name does not contain a slash, it is interpreted as an environment variable name, which must be known to the server process. This way the database administrator can exercise control over locations in which databases can be created. (A customary choice is, e.g., `PGDATA2`.) If the server is compiled with `ALLOW_ABSOLUTE_DBPATHS` (not so by default), absolute path names, as identified by a leading slash (e.g., `/usr/local/pgsql/data`), are allowed as well.

By default, the new database will be created by cloning the standard system database `template1`. A different template can be specified by writing `TEMPLATE = name`. In particular, by writing `TEMPLATE = template0`, you can create a virgin database containing only the standard objects predefined by your version of PostgreSQL. This is useful if you wish to avoid copying any installation-local objects that may have been added to `template1`.

The optional encoding parameter allows selection of the database encoding, if your server was compiled with multibyte encoding support. When not specified, it defaults to the encoding used by the selected template database.

Optional parameters can be written in any order, not only the order illustrated above.

Notes

`CREATE DATABASE` is a PostgreSQL language extension.

Use DROP DATABASE to remove a database.

The program createdb is a shell script wrapper around this command, provided for convenience.

There are security and data integrity issues involved with using alternate database locations specified with absolute path names, and by default only an environment variable known to the backend may be specified for an alternate location. See the Administrator's Guide for more information.

Although it is possible to copy a database other than `template1` by specifying its name as the template, this is not (yet) intended as a general-purpose COPY DATABASE facility. We recommend that databases used as templates be treated as read-only. See the *Administrator's Guide* for more information.

Usage

To create a new database:

```
olly=> create database lusiadas;
```

To create a new database in an alternate area `~/private_db`:

```
$ mkdir private_db
$ initlocation ~/private_db
    The location will be initialized with username "olly".
This user will own all the files and must also own the server process.
Creating directory /home/olly/private_db
Creating directory /home/olly/private_db/base

initlocation is complete.


$ psql olly
Welcome to psql, 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

olly=> CREATE DATABASE elsewhere WITH LOCATION = '/home/olly/
private_db';
CREATE DATABASE
```

Compatibility

SQL92

There is no `CREATE DATABASE` statement in SQL92. Databases are equivalent to catalogs whose creation is implementation-defined.

Name

CREATE FUNCTION — define a new function

Synopsis

```
CREATE [ OR REPLACE ] FUNCTION name ( [ argtype [, ...] ] )
    RETURNS rettype
    AS 'definition'
    LANGUAGE langname
    [ WITH ( attribute [, ...] ) ]
CREATE [ OR REPLACE ] FUNCTION name ( [ argtype [, ...] ] )
    RETURNS rettype
    AS 'obj_file', 'link_symbol'
    LANGUAGE langname
    [ WITH ( attribute [, ...] ) ]
```

Description

`CREATE FUNCTION` defines a new function. `CREATE OR REPLACE FUNCTION` will either create a new function, or replace an existing definition.

Parameters

name

The name of a function to create. The name need not be unique, because functions may be overloaded, but functions with the same name must have different argument types.

argtype

The data type(s) of the function's arguments, if any. The input types may be base or complex types, `opaque`, or the same as the type of an existing column. `Opaque` indicates that the function accepts arguments of a non-SQL type such as `char *`. The type of a column is indicated using *tablename.columnname*`%TYPE`; using this can sometimes help make a function independent from changes to the definition of a table.

rettype

The return data type. The output type may be specified as a base type, complex type, `setof` type, `opaque`, or the same as the type of an existing column. The `setof` modifier indicates that the function will return a set of items, rather than a single item. Functions with a declared return type of `opaque` do not return a value. These cannot be called directly; trigger functions make use of this feature.

definition

A string defining the function; the meaning depends on the language. It may be an internal function name, the path to an object file, an SQL query, or text in a procedural language.

obj_file, *link_symbol*

This form of the `AS` clause is used for dynamically linked C language functions when the function name in the C language source code is not the same as the name of the SQL function. The string

obj_file is the name of the file containing the dynamically loadable object, and *link_symbol* is the object's link symbol, that is, the name of the function in the C language source code.

langname

May be SQL, C, internal, or *plname*, where *plname* is the name of a created procedural language. See CREATE LANGUAGE for details. For backward compatibility, the name may be enclosed by single quotes.

attribute

An optional piece of information about the function, used for optimization. See below for details.

The user that creates the function becomes the owner of the function.

The following attributes may appear in the WITH clause:

iscachable

Iscachable indicates that the function always returns the same result when given the same argument values (i.e., it does not do database lookups or otherwise use information not directly present in its parameter list). The optimizer uses iscachable to know whether it is safe to pre-evaluate a call of the function.

isstrict

isstrict indicates that the function always returns NULL whenever any of its arguments are NULL. If this attribute is specified, the function is not executed when there are NULL arguments; instead a NULL result is assumed automatically. When isstrict is not specified, the function will be called for NULL inputs. It is then the function author's responsibility to check for NULLs if necessary and respond appropriately.

Notes

Refer to the chapter in the *PostgreSQL Programmer's Guide* on the topic of extending PostgreSQL via functions for further information on writing external functions.

The full SQL type syntax is allowed for input arguments and return value. However, some details of the type specification (e.g., the precision field for numeric types) are the responsibility of the underlying function implementation and are silently swallowed (i.e., not recognized or enforced) by the CREATE FUNCTION command.

PostgreSQL allows function *overloading*; that is, the same name can be used for several different functions so long as they have distinct argument types. This facility must be used with caution for internal and C-language functions, however.

Two internal functions cannot have the same C name without causing errors at link time. To get around that, give them different C names (for example, use the argument types as part of the C names), then specify those names in the AS clause of CREATE FUNCTION. If the AS clause is left empty, then CREATE FUNCTION assumes the C name of the function is the same as the SQL name.

Similarly, when overloading SQL function names with multiple C-language functions, give each C-language instance of the function a distinct name, then use the alternative form of the AS clause in the CREATE FUNCTION syntax to select the appropriate C-language implementation of each overloaded SQL function.

When repeated `CREATE FUNCTION` calls refer to the same object file, the file is only loaded once. To unload and reload the file (perhaps during development), use the LOAD command.

Use `DROP FUNCTION` to remove user-defined functions.

To update the definition of an existing function, use `CREATE OR REPLACE FUNCTION`. Note that it is not possible to change the name or argument types of a function this way (if you tried, you'd just be creating a new, distinct function). Also, `CREATE OR REPLACE FUNCTION` will not let you change the return type of an existing function. To do that, you must drop and re-create the function.

If you drop and then re-create a function, the new function is not the same entity as the old; you will break existing rules, views, triggers, etc that referred to the old function. Use `CREATE OR REPLACE FUNCTION` to change a function definition without breaking objects that refer to the function.

Examples

To create a simple SQL function:

```
CREATE FUNCTION one() RETURNS integer
    AS 'SELECT 1 AS RESULT;'
    LANGUAGE SQL;

SELECT one() AS answer;
 answer
--------
      1
```

The next example creates a C function by calling a routine from a user-created shared library named `funcs.so` (the extension may vary across platforms). The shared library file is sought in the server's dynamic library search path. This particular routine calculates a check digit and returns TRUE if the check digit in the function parameters is correct. It is intended for use in a CHECK constraint.

```
CREATE FUNCTION ean_checkdigit(char, char) RETURNS boolean
    AS 'funcs' LANGUAGE C;

CREATE TABLE product (
    id        char(8) PRIMARY KEY,
    eanprefix char(8) CHECK (eanprefix ~ '[0-9]{2}-[0-9]{5}')
                      REFERENCES brandname(ean_prefix),
    eancode   char(6) CHECK (eancode ~ '[0-9]{6}'),
    CONSTRAINT ean    CHECK (ean_checkdigit(eanprefix, eancode))
);
```

This example creates a function that does type conversion between the user-defined type complex, and the internal type point. The function is implemented by a dynamically loaded object that was compiled from C source (we illustrate the now-deprecated alternative of specifying the exact pathname to the shared object file). For PostgreSQL to find a type conversion function automatically, the SQL function has to have the same name as the return type, and so overloading is unavoidable. The function name is overloaded by using the second form of the `AS` clause in the SQL definition:

```
CREATE FUNCTION point(complex) RETURNS point
    AS '/home/bernie/pgsql/lib/complex.so', 'complex_to_point'
    LANGUAGE C;
```

The C declaration of the function could be:

```
Point * complex_to_point (Complex *z)
{
 Point *p;

 p = (Point *) palloc(sizeof(Point));
 p->x = z->x;
 p->y = z->y;

 return p;
}
```

Compatibility

A `CREATE FUNCTION` command is defined in SQL99. The PostgreSQL version is similar but not compatible. The attributes are not portable, neither are the different available languages.

See Also

DROP FUNCTION , LOAD, *PostgreSQL Programmer's Guide*

Name

CREATE GROUP — define a new user group

Synopsis

<refsynopsisdivinfo>2000-01-14</refsynopsisdivinfo>

```
CREATE GROUP name [ [ WITH ] option [ ... ] ]

where option can be:

      SYSID gid
    | USER  username [, ...]
```

Inputs

name

The name of the group.

gid

The SYSID clause can be used to choose the PostgreSQL group id of the new group. It is not necessary to do so, however.

If this is not specified, the highest assigned group id plus one, starting at 1, will be used as default.

username

A list of users to include in the group. The users must already exist.

Outputs

CREATE GROUP

Message returned if the command completes successfully.

Description

CREATE GROUP will create a new group in the database installation. Refer to the *Administrator's Guide* for information about using groups for authentication. You must be a database superuser to use this command.

Use ALTER GROUP to change a group's membership, and DROP GROUP to remove a group.

Usage

Create an empty group:

```
CREATE GROUP staff;
```

Create a group with members:

```
CREATE GROUP marketing WITH USER jonathan, david;
```

Compatibility

SQL92

There is no `CREATE GROUP` statement in SQL92. Roles are similar in concept to groups.

Name

CREATE INDEX — define a new index

Synopsis

<refsynopsisdivinfo>2001-07-15</refsynopsisdivinfo>

```
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_method ] ( column [ ops_name ] [, ...] )
    [ WHERE predicate ]
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_method ] ( func_name( column [, ... ]) [ ops_name ] )
    [ WHERE predicate ]
```

Inputs

UNIQUE

Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error.

index_name

The name of the index to be created.

table

The name of the table to be indexed.

acc_method

The name of the access method to be used for the index. The default access method is `BTREE`. PostgreSQL provides four access methods for indexes:

BTREE

an implementation of Lehman-Yao high-concurrency B-trees.

RTREE

implements standard R-trees using Guttman's quadratic split algorithm.

HASH

an implementation of Litwin's linear hashing.

GIST

Generalized Index Search Trees.

column

The name of a column of the table.

ops_name

An associated operator class. See below for details.

func_name

A function, which returns a value that can be indexed.

predicate

Defines the constraint expression for a partial index.

Outputs

CREATE

The message returned if the index is successfully created.

ERROR: Cannot create index: 'index_name' already exists.

This error occurs if it is impossible to create the index.

Description

CREATE INDEX constructs an index *index_name* on the specified *table*.

Tip

Indexes are primarily used to enhance database performance. But inappropriate use will result in slower performance.

In the first syntax shown above, the key field(s) for the index are specified as column names. Multiple fields can be specified if the index access method supports multicolumn indexes.

In the second syntax shown above, an index is defined on the result of a user-specified function *func_name* applied to one or more columns of a single table. These *functional indexes* can be used to obtain fast access to data based on operators that would normally require some transformation to apply them to the base data.

PostgreSQL provides B-tree, R-tree, hash, and GiST access methods for indexes. The B-tree access method is an implementation of Lehman-Yao high-concurrency B-trees. The R-tree access method implements standard R-trees using Guttman's quadratic split algorithm. The hash access method is an implementation of Litwin's linear hashing. We mention the algorithms used solely to indicate that all of these access methods are fully dynamic and do not have to be optimized periodically (as is the case with, for example, static hash access methods).

When the WHERE clause is present, a *partial index* is created. A partial index is an index that contains entries for only a portion of a table, usually a portion that is somehow more interesting than the rest of the table. For example, if you have a table that contains both billed and unbilled orders where the unbilled orders take up a small fraction of the total table and yet that is an often used section, you can improve performance by creating an index on just that portion. Another possible application is to use WHERE with UNIQUE to enforce uniqueness over a subset of a table.

The expression used in the WHERE clause may refer only to columns of the underlying table (but it can use all columns, not only the one(s) being indexed). Presently, sub-SELECTs and aggregate expressions are also forbidden in WHERE.

All functions and operators used in an index definition must be *cachable*, that is, their results must depend only on their input arguments and never on any outside influence (such as the contents of another table or the current time). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index, remember to mark the function cachable when you create it.

Use DROP INDEX to remove an index.

Notes

The PostgreSQL query optimizer will consider using a B-tree index whenever an indexed attribute is involved in a comparison using one of: <, <=, =, >=, >

The PostgreSQL query optimizer will consider using an R-tree index whenever an indexed attribute is involved in a comparison using one of: <<, &<, &>, >>, @, ~=, &&

The PostgreSQL query optimizer will consider using a hash index whenever an indexed attribute is involved in a comparison using the = operator.

Currently, only the B-tree and gist access methods support multi-column indexes. Up to 16 keys may be specified by default (this limit can be altered when building PostgreSQL). Only B-tree currently supports unique indexes.

An *operator class* can be specified for each column of an index. The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on four-byte integers would use the `int4_ops` class; this operator class includes comparison functions for four-byte integers. In practice the default operator class for the field's data type is usually sufficient. The main point of having operator classes is that for some data types, there could be more than one meaningful ordering. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index. There are also some operator classes with special purposes:

- The operator classes `box_ops` and `bigbox_ops` both support R-tree indexes on the `box` data type. The difference between them is that `bigbox_ops` scales box coordinates down, to avoid floating-point exceptions from doing multiplication, addition, and subtraction on very large floating-point coordinates. (Note: this was true some time ago, but currently the two operator classes both use floating point and are effectively identical.)

The following query shows all defined operator classes:

```
SELECT am.amname AS acc_method,
       opc.opcname AS ops_name,
       opr.oprname AS ops_comp
    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 acc_method, ops_name, ops_comp;
```

Usage

To create a B-tree index on the field `title` in the table `films`:

```
CREATE UNIQUE INDEX title_idx
    ON films (title);
```

Compatibility

SQL92

CREATE INDEX is a PostgreSQL language extension.

There is no `CREATE INDEX` command in SQL92.

<docinfo>2001-09-05</docinfo>

Name

CREATE LANGUAGE — define a new procedural language

Synopsis

```
CREATE [ TRUSTED ] [ PROCEDURAL ] LANGUAGE langname
    HANDLER call_handler
```

Description

Using `CREATE LANGUAGE`, a PostgreSQL user can register a new procedural language with a PostgreSQL database. Subsequently, functions and trigger procedures can be defined in this new language. The user must have the PostgreSQL superuser privilege to register a new language.

`CREATE LANGUAGE` effectively associates the language name with a call handler that is responsible for executing functions written in the language. Refer to the *Programmer's Guide* for more information about language call handlers.

Note that procedural languages are local to individual databases. To make a language available in all databases by default, it should be installed into the `template1` database.

Parameters

`TRUSTED`

`TRUSTED` specifies that the call handler for the language is safe, that is, it does not offer an unprivileged user any functionality to bypass access restrictions. If this keyword is omitted when registering the language, only users with the PostgreSQL superuser privilege can use this language to create new functions.

`PROCEDURAL`

This is a noise word.

`langname`

The name of the new procedural language. The language name is case insensitive. A procedural language cannot override one of the built-in languages of PostgreSQL.

For backward compatibility, the name may be enclosed by single quotes.

`HANDLER` *`call_handler`*

`call_handler` is the name of a previously registered function that will be called to execute the procedural language functions. The call handler for a procedural language must be written in a compiled language such as C with version 1 call convention and registered with PostgreSQL as a function taking no arguments and returning the `opaque` type, a placeholder for unspecified or undefined types.

Diagnostics

`CREATE`

This message is returned if the language is successfully created.

```
ERROR:  PL handler function funcname() doesn't exist
```

This error is returned if the function *funcname*() is not found.

Notes

This command normally should not be executed directly by users. For the procedural languages supplied in the PostgreSQL distribution, the createlang script should be used, which will also install the correct call handler. (`createlang` will call `CREATE LANGUAGE` internally.)

Use the CREATE FUNCTION command to create a new function.

Use DROP LANGUAGE, or better yet the droplang script, to drop procedural languages.

The system catalog `pg_language` records information about the currently installed procedural languages.

```
        Table "pg_language"
   Attribute   |  Type   | Modifier
---------------+---------+----------
 lanname       | name    |
 lanispl       | boolean |
 lanpltrusted  | boolean |
 lanplcallfoid | oid     |
 lancompiler   | text    |

   lanname   | lanispl | lanpltrusted | lanplcallfoid | lancompiler
-------------+---------+--------------+---------------+-------------
 internal    | f       | f            |             0 | n/a
 C           | f       | f            |             0 | /bin/cc
 sql         | f       | f            |             0 | postgres
```

At present, the definition of a procedural language cannot be changed once it has been created.

Examples

The following two commands executed in sequence will register a new procedural language and the associated call handler.

```
CREATE FUNCTION plsample_call_handler () RETURNS opaque
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

Compatibility

`CREATE LANGUAGE` is a PostgreSQL extension.

History

The `CREATE LANGUAGE` command first appeared in PostgreSQL 6.3.

See Also

createlang, CREATE FUNCTION, droplang, DROP LANGUAGE, *PostgreSQL Programmer's Guide*

Name

CREATE OPERATOR — define a new operator

Synopsis

<refsynopsisdivinfo>2000-03-25</refsynopsisdivinfo>

```
CREATE OPERATOR name ( PROCEDURE = func_name
     [, LEFTARG = lefttype
     ] [, RIGHTARG = righttype ]
     [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
     [, RESTRICT = res_proc ] [, JOIN = join_proc ]
     [, HASHES ] [, SORT1 = left_sort_op ] [, SORT2 = right_sort_op
 ] )
```

Inputs

name

The operator to be defined. See below for allowable characters.

func_name

The function used to implement this operator.

lefttype

The type of the left-hand argument of the operator, if any. This option would be omitted for a left-unary operator.

righttype

The type of the right-hand argument of the operator, if any. This option would be omitted for a right-unary operator.

com_op

The commutator of this operator.

neg_op

The negator of this operator.

res_proc

The restriction selectivity estimator function for this operator.

join_proc

The join selectivity estimator function for this operator.

HASHES

Indicates this operator can support a hash join.

left_sort_op

If this operator can support a merge join, the operator that sorts the left-hand data type of this operator.

right_sort_op

If this operator can support a merge join, the operator that sorts the right-hand data type of this operator.

Outputs

CREATE

Message returned if the operator is successfully created.

Description

CREATE OPERATOR defines a new operator, *name*. The user who defines an operator becomes its owner.

The operator *name* is a sequence of up to NAMEDATALEN-1 (31 by default) characters from the following list:

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

There are a few restrictions on your choice of name:

- $ cannot be defined as a single-character operator, although it can be part of a multicharacter operator name.
- -- and /* cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
- A multicharacter operator name cannot end in + or -, unless the name also contains at least one of these characters:

  ~ ! @ # % ^ & | ` ? $

  For example, @- is an allowed operator name, but *- is not. This restriction allows PostgreSQL to parse SQL-compliant queries without requiring spaces between tokens.

  ### Note

  When working with non-SQL-standard operator names, you will usually need to separate adjacent operators with spaces to avoid ambiguity. For example, if you have defined a left-unary operator named @, you cannot write X*@Y; you must write X* @Y to ensure that PostgreSQL reads it as two operator names not one.

The operator != is mapped to <> on input, so these two names are always equivalent.

At least one of LEFTARG and RIGHTARG must be defined. For binary operators, both should be defined. For right unary operators, only LEFTARG should be defined, while for left unary operators only RIGHTARG should be defined.

The *func_name* procedure must have been previously defined using CREATE FUNCTION and must be defined to accept the correct number of arguments (either one or two) of the indicated types.

The commutator operator should be identified if one exists, so that PostgreSQL can reverse the order of the operands if it wishes. For example, the operator area-less-than, <<<, would probably have a commutator operator, area-greater-than, >>>. Hence, the query optimizer could freely convert:

```
box '((0,0), (1,1))'  >>> MYBOXES.description
```

to

```
MYBOXES.description <<< box '((0,0), (1,1))'
```

This allows the execution code to always use the latter representation and simplifies the query optimizer somewhat.

Similarly, if there is a negator operator then it should be identified. Suppose that an operator, area-equal, ===, exists, as well as an area not equal, !==. The negator link allows the query optimizer to simplify

```
NOT MYBOXES.description === box '((0,0), (1,1))'
```

to

```
MYBOXES.description !== box '((0,0), (1,1))'
```

If a commutator operator name is supplied, PostgreSQL searches for it in the catalog. If it is found and it does not yet have a commutator itself, then the commutator's entry is updated to have the newly created operator as its commutator. This applies to the negator, as well. This is to allow the definition of two operators that are the commutators or the negators of each other. The first operator should be defined without a commutator or negator (as appropriate). When the second operator is defined, name the first as the commutator or negator. The first will be updated as a side effect. (As of PostgreSQL `6.5`, it also works to just have both operators refer to each other.)

The HASHES, SORT1, and SORT2 options are present to support the query optimizer in performing joins. PostgreSQL can always evaluate a join (i.e., processing a clause with two tuple variables separated by an operator that returns a `boolean`) by iterative substitution [WONG76]. In addition, PostgreSQL can use a hash-join algorithm along the lines of [SHAP86]; however, it must know whether this strategy is applicable. The current hash-join algorithm is only correct for operators that represent equality tests; furthermore, equality of the data type must mean bitwise equality of the representation of the type. (For example, a data type that contains unused bits that don't matter for equality tests could not be hashjoined.) The HASHES flag indicates to the query optimizer that a hash join may safely be used with this operator.

Similarly, the two sort operators indicate to the query optimizer whether merge-sort is a usable join strategy and which operators should be used to sort the two operand classes. Sort operators should only be provided for an equality operator, and they should refer to less-than operators for the left and right side data types respectively.

If other join strategies are found to be practical, PostgreSQL will change the optimizer and run-time system to use them and will require additional specification when an operator is defined. Fortunately, the research community invents new join strategies infrequently, and the added generality of user-defined join strategies was not felt to be worth the complexity involved.

The RESTRICT and JOIN options assist the query optimizer in estimating result sizes. If a clause of the form:

```
MYBOXES.description <<< box '((0,0), (1,1))'
```

is present in the qualification, then PostgreSQL may have to estimate the fraction of the instances in MYBOXES that satisfy the clause. The function *res_proc* must be a registered function (meaning it is already defined using CREATE FUNCTION) which accepts arguments of the correct data types and returns a floating-point number. The query optimizer simply calls this function, passing the parameter ((0,0), (1,1)) and multiplies the result by the relation size to get the expected number of instances.

Similarly, when the operands of the operator both contain instance variables, the query optimizer must estimate the size of the resulting join. The function join_proc will return another floating-point number which will be multiplied by the cardinalities of the two tables involved to compute the expected result size.

The difference between the function

```
my_procedure_1 (MYBOXES.description, box '((0,0), (1,1))')
```

and the operator

```
MYBOXES.description === box '((0,0), (1,1))'
```

is that PostgreSQL attempts to optimize operators and can decide to use an index to restrict the search space when operators are involved. However, there is no attempt to optimize functions, and they are performed by brute force. Moreover, functions can have any number of arguments while operators are restricted to one or two.

Notes

Refer to the chapter on operators in the *PostgreSQL User's Guide* for further information. Refer to DROP OPERATOR to delete user-defined operators from a database.

Usage

The following command defines a new operator, area-equality, for the BOX data type:

```
CREATE OPERATOR === (
   LEFTARG = box,
   RIGHTARG = box,
   PROCEDURE = area_equal_procedure,
   COMMUTATOR = ===,
   NEGATOR = !==,
   RESTRICT = area_restriction_procedure,
   JOIN = area_join_procedure,
   HASHES,
   SORT1 = <<<,
   SORT2 = <<<
);
```

Compatibility

SQL92

CREATE OPERATOR is a PostgreSQL extension. There is no CREATE OPERATOR statement in SQL92.

Name

CREATE RULE — define a new rewrite rule

Synopsis

<refsynopsisdivinfo>2001-01-05</refsynopsisdivinfo>

```
CREATE RULE name AS ON event
    TO object [ WHERE condition ]
    DO [ INSTEAD ] action

where action can be:

NOTHING
|
query
|
( query ; query ... )
|
[ query ; query ... ]
```

Inputs

name

The name of a rule to create.

event

Event is one of SELECT, UPDATE, DELETE or INSERT.

object

Object is either *table* or *table.column*. (Currently, only the *table* form is actually implemented.)

condition

Any SQL boolean-condition expression. The condition expression may not refer to any tables except new and old.

query

The query or queries making up the *action* can be any SQL SELECT, INSERT, UPDATE, DELETE, or NOTIFY statement.

Within the *condition* and *action*, the special table names new and old may be used to refer to values in the referenced table (the *object*). new is valid in ON INSERT and ON UPDATE rules to refer to the new row being inserted or updated. old is valid in ON UPDATE and ON DELETE rules to refer to the existing row being updated or deleted.

Outputs

CREATE

Message returned if the rule is successfully created.

Description

The PostgreSQL *rule system* allows one to define an alternate action to be performed on inserts, updates, or deletions from database tables. Rules are used to implement table views as well.

The semantics of a rule is that at the time an individual instance (row) is accessed, inserted, updated, or deleted, there is an old instance (for selects, updates and deletes) and a new instance (for inserts and updates). All the rules for the given event type and the given target object (table) are examined, in an unspecified order. If the *condition* specified in the WHERE clause (if any) is true, the *action* part of the rule is executed. The *action* is done instead of the original query if INSTEAD is specified; otherwise it is done after the original query in the case of ON INSERT, or before the original query in the case of ON UPDATE or ON DELETE. Within both the *condition* and *action*, values from fields in the old instance and/or the new instance are substituted for `old.`*attribute-name* and `new.`*attribute-name*.

The *action* part of the rule can consist of one or more queries. To write multiple queries, surround them with either parentheses or square brackets. Such queries will be performed in the specified order (whereas there are no guarantees about the execution order of multiple rules for an object). The *action* can also be NOTHING indicating no action. Thus, a DO INSTEAD NOTHING rule suppresses the original query from executing (when its condition is true); a DO NOTHING rule is useless.

The *action* part of the rule executes with the same command and transaction identifier as the user command that caused activation.

Rules and Views

Presently, ON SELECT rules must be unconditional INSTEAD rules and must have actions that consist of a single SELECT query. Thus, an ON SELECT rule effectively turns the object table into a view, whose visible contents are the rows returned by the rule's SELECT query rather than whatever had been stored in the table (if anything). It is considered better style to write a CREATE VIEW command than to create a real table and define an ON SELECT rule for it.

CREATE VIEW creates a dummy table (with no underlying storage) and associates an ON SELECT rule with it. The system will not allow updates to the view, since it knows there is no real table there. You can create the illusion of an updatable view by defining ON INSERT, ON UPDATE, and ON DELETE rules (or any subset of those that's sufficient for your purposes) to replace update actions on the view with appropriate updates on other tables.

There is a catch if you try to use conditional rules for view updates: there *must* be an unconditional INSTEAD rule for each action you wish to allow on the view. If the rule is conditional, or is not INSTEAD, then the system will still reject attempts to perform the update action, because it thinks it might end up trying to perform the action on the dummy table in some cases. If you want to handle all the useful cases in conditional rules, you can; just add an unconditional DO INSTEAD NOTHING rule to ensure that the system understands it will never be called on to update the dummy table. Then make the conditional rules non-INSTEAD; in the cases where they fire, they add to the default INSTEAD NOTHING action.

Notes

You must have rule definition access to a table in order to define a rule on it. Use `GRANT` and `REVOKE` to change permissions.

It is very important to take care to avoid circular rules. For example, though each of the following two rule definitions are accepted by PostgreSQL, the select command will cause PostgreSQL to report an error because the query cycled too many times:

```
CREATE RULE "_RETemp" AS
    ON SELECT TO emp
    DO INSTEAD
 SELECT * FROM toyemp;

CREATE RULE "_RETtoyemp" AS
    ON SELECT TO toyemp
    DO INSTEAD
 SELECT * FROM emp;
```

This attempt to select from EMP will cause PostgreSQL to issue an error because the queries cycled too many times:

```
SELECT * FROM emp;
```

Presently, if a rule contains a NOTIFY query, the NOTIFY will be executed unconditionally --- that is, the NOTIFY will be issued even if there are not any rows that the rule should apply to. For example, in

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;
```

one NOTIFY event will be sent during the UPDATE, whether or not there are any rows with id = 42. This is an implementation restriction that may be fixed in future releases.

Compatibility

SQL92

`CREATE RULE` statement is a PostgreSQL language extension. There is no `CREATE RULE` statement in SQL92.

Name

CREATE SEQUENCE — define a new sequence generator

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE seqname [ INCREMENT increment ]
    [ MINVALUE minvalue ] [ MAXVALUE maxvalue ]
    [ START start ] [ CACHE cache ] [ CYCLE ]
```

Inputs

TEMPORARY or TEMP

If specified, the sequence object is created only for this session, and is automatically dropped on session exit. Existing permanent sequences with the same name are not visible (in this session) while the temporary sequence exists.

seqname

The name of a sequence to be created.

increment

The `INCREMENT` *increment* clause is optional. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is one (1).

minvalue

The optional clause `MINVALUE` *minvalue* determines the minimum value a sequence can generate. The defaults are 1 and -2^63-1 for ascending and descending sequences, respectively.

maxvalue

The optional clause `MAXVALUE` *maxvalue* determines the maximum value for the sequence. The defaults are 2^63-1 and -1 for ascending and descending sequences, respectively.

start

The optional `START` *start* `clause` enables the sequence to begin anywhere. The default starting value is *minvalue* for ascending sequences and *maxvalue* for descending ones.

cache

The `CACHE` *cache* option enables sequence numbers to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., no cache) and this is also the default.

CYCLE

The optional CYCLE keyword may be used to enable the sequence to wrap around when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the *minvalue* or *maxvalue*, respectively. Without CYCLE, after the limit is reached `nextval` calls will return an error.

Outputs

`CREATE`

Message returned if the command is successful.

`ERROR: Relation '`*`seqname`*`' already exists`

If the sequence specified already exists.

`ERROR: DefineSequence: MINVALUE (`*`start`*`) can't be >= MAXVALUE (`*`max`*`)`

If the specified starting value is out of range.

`ERROR: DefineSequence: START value (`*`start`*`) can't be < MINVALUE (`*`min`*`)`

If the specified starting value is out of range.

`ERROR: DefineSequence: MINVALUE (`*`min`*`) can't be >= MAXVALUE (`*`max`*`)`

If the minimum and maximum values are inconsistent.

Description

`CREATE SEQUENCE` will enter a new sequence number generator into the current database. This involves creating and initializing a new single-row table with the name *`seqname`*. The generator will be owned by the user issuing the command.

After a sequence is created, you use the functions `nextval`, `currval` and `setval` to operate on the sequence. These functions are documented in the *User's Guide*.

Although you cannot update a sequence directly, you can use a query like

```
SELECT * FROM seqname;
```

to examine the parameters and current state of a sequence. In particular, the `last_value` field of the sequence shows the last value allocated by any backend process. (Of course, this value may be obsolete by the time it's printed, if other processes are actively doing `nextval` calls.)

Caution

Unexpected results may be obtained if a *`cache`* setting greater than one is used for a sequence object that will be used concurrently by multiple backends. Each backend will allocate and cache successive sequence values during one access to the sequence object and increase the sequence object's `last_value` accordingly. Then, the next *`cache`*-1 uses of `nextval` within that backend simply return the preallocated values without touching the shared object. So, any numbers allocated but not used within a session will be lost when that session ends. Furthermore, although multiple backends are guaranteed to allocate distinct sequence values, the values may be generated out of sequence when all the backends are considered. (For example, with a *`cache`* setting of 10, backend A might reserve values 1..10 and return `nextval`=1, then backend B might reserve values 11..20 and return `nextval`=11 before backend A has generated nextval=2.) Thus, with a *`cache`* setting of one it is safe to assume that `nextval` values are generated sequentially; with a *`cache`* setting greater than one you should only assume that the `nextval` values are all distinct, not that they are generated purely sequentially. Also, `last_value` will

reflect the latest value reserved by any backend, whether or not it has yet been returned by `nextval`. Another consideration is that a `setval` executed on such a sequence will not be noticed by other backends until they have used up any preallocated values they have cached.

Notes

Use `DROP SEQUENCE` to remove a sequence.

Sequences are based on `bigint` arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807). On some older platforms, there may be no compiler support for eight-byte integers, in which case sequences use regular `integer` arithmetic (range -2147483648 to +2147483647).

When *`cache`* is greater than one, each backend uses its own cache to store preallocated numbers. Numbers that are cached but not used in the current session will be lost, resulting in "holes" in the sequence.

Usage

Create an ascending sequence called `serial`, starting at 101:

```
CREATE SEQUENCE serial START 101;
```

Select the next number from this sequence:

```
SELECT nextval('serial');

nextval
-------
    114
```

Use this sequence in an INSERT:

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

Update the sequence value after a COPY FROM:

```
BEGIN;
    COPY distributors FROM 'input_file';
    SELECT setval('serial', max(id)) FROM distributors;
END;
```

Compatibility

SQL92

`CREATE SEQUENCE` is a PostgreSQL language extension. There is no `CREATE SEQUENCE` statement in SQL92.

Name

CREATE TABLE — define a new table

Synopsis

```
CREATE [ [ LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name (
    { column_name data_type [ DEFAULT default_expr ]
 [ column_constraint [, ... ] ]
    | table_constraint }  [, ... ]
)
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH OIDS | WITHOUT OIDS ]

where column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | UNIQUE | PRIMARY KEY |
  CHECK (expression) |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY
 IMMEDIATE ]

and table_constraint is:

[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ... ] ) |
  PRIMARY KEY ( column_name [, ... ] ) |
  CHECK ( expression ) |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable
 [ ( refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL ] [ ON DELETE action ] [ ON
 UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY
 IMMEDIATE ]
```

Description

CREATE TABLE will create a new, initially empty table in the current database. The table will be owned by the user issuing the command.

CREATE TABLE also automatically creates a data type that represents the tuple type (structure type) corresponding to one row of the table. Therefore, tables cannot have the same name as any existing data type.

A table cannot have more than 1600 columns. (In practice, the effective limit is lower because of tuple-length constraints). A table cannot have the same name as a system catalog table.

The optional constraint clauses specify constraints (or tests) that new or updated rows must satisfy for an insert or update operation to succeed. A constraint is a named rule: an SQL object which helps define valid sets of values by putting limits on the results of insert, update, or delete operations performed on a table.

There are two ways to define constraints: table constraints and column constraints. A column constraint is defined as part of a column definition. A table constraint definition is not tied to a particular column, and

it can encompass more than one column. Every column constraint can also be written as a table constraint; a column constraint is only a notational convenience if the constraint only affects one column.

Parameters

`[LOCAL] TEMPORARY` or `[LOCAL] TEMP`

If specified, the table is created as a temporary table. Temporary tables are automatically dropped at the end of a session. Existing persistent tables with the same name are not visible to the current session while the temporary table exists. Any indexes created on a temporary table are automatically temporary as well.

The `LOCAL` word is optional. But see under Compatibility.

`table_name`

The name of the table to be created.

`column_name`

The name of a column to be created in the new table.

`data_type`

The data type of the column. This may include array specifiers. Refer to the *User's Guide* for further information about data types and arrays.

`DEFAULT` *`default_expr`*

The `DEFAULT` clause assigns a default data value for the column whose column definition it appears within. The value is any variable-free expression (subselects and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column.

The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is NULL.

`INHERITS ( parent_table [, ... ] )`

The optional `INHERITS` clause specifies a list of tables from which the new table automatically inherits all columns. If the same column name exists in more than one parent table, an error is reported unless the data types of the columns match in each of the parent tables. If there is no conflict, then the duplicate columns are merged to form a single column in the new table. If the column name list of the new table contains a column that is also inherited, the data type must likewise match the inherited column(s), and the column definitions are merged into one. However, inherited and new column declarations of the same name need not specify identical constraints: all constraints provided from any declaration are merged together and all are applied to the new table. If the new table explicitly specifies a default value for the column, this default overrides any defaults from inherited declarations of the column. Otherwise, any parents that specify default values for the column must all specify the same default, or an error will be reported.

`WITH OIDS` or `WITHOUT OIDS`

This optional clause specifies whether rows of the new table should have OIDs (object identifiers) assigned to them. The default is to have OIDs. (If the new table inherits from any tables that have OIDs, then `WITH OIDS` is forced even if the command says `WITHOUT OIDS`.)

Specifying WITHOUT OIDS allows the user to suppress generation of OIDs for rows of a table. This may be worthwhile for large tables, since it will reduce OID consumption and thereby postpone wraparound of the 32-bit OID counter. Once the counter wraps around, uniqueness of OIDs can no longer be assumed, which considerably reduces their usefulness.

CONSTRAINT *constraint_name*

An optional name for a column or table constraint. If not specified, the system generates a name.

NOT NULL

The column is not allowed to contain NULL values. This is equivalent to the column constraint CHECK (*column* NOT NULL).

NULL

The column is allowed to contain NULL values. This is the default.

This clause is only available for compatibility with non-standard SQL databases. Its use is discouraged in new applications.

UNIQUE (column constraint)
UNIQUE (*column_name* [, ...]) (table constraint)

The UNIQUE constraint specifies a rule that a group of one or more distinct columns of a table may contain only unique values. The behavior of the unique table constraint is the same as that for column constraints, with the additional capability to span multiple columns.

For the purpose of a unique constraint, NULL values are not considered equal.

Each unique table constraint must name a set of columns that is different from the set of columns named by any other unique or primary key constraint defined for the table. (Otherwise it would just be the same constraint listed twice.)

PRIMARY KEY (column constraint)
PRIMARY KEY (*column_name* [, ...]) (table constraint)

The primary key constraint specifies that a column or columns of a table may contain only unique (non-duplicate), non-NULL values. Technically, PRIMARY KEY is merely a combination of UNIQUE and NOT NULL, but identifying a set of columns as primary key also provides meta-data about the design of the schema, as a primary key implies that other tables may rely on this set of columns as a unique identifier for rows.

Only one primary key can be specified for a table, whether as a column constraint or a table constraint.

The primary key constraint should name a set of columns that is different from other sets of columns named by any unique constraint defined for the same table.

CHECK (*expression*)

CHECK clauses specify integrity constraints or tests which new or updated rows must satisfy for an insert or update operation to succeed. Each constraint must be an expression producing a Boolean result. A condition appearing within a column definition should reference that column's value only, while a condition appearing as a table constraint may reference multiple columns.

Currently, CHECK expressions cannot contain subselects nor refer to variables other than columns of the current row.

`REFERENCES` *`reftable`* `[ ( ` *`refcolumn`* ` ) ] [ MATCH` *`matchtype`* `] [ ON DELETE` *`action`* `] [ ON UPDATE` *`action`* `]` (column constraint)
`FOREIGN KEY ( ` *`column`* `[, ... ] ) REFERENCES` *`reftable`* `[ ( ` *`refcolumn`* `[, ... ] ) ] [ MATCH` *`matchtype`* `] [ ON DELETE` *`action`* `] [ ON UPDATE` *`action`* `]` (table constraint)

The `REFERENCES` column constraint specifies that a group of one or more columns of the new table must only contain values which match against values in the referenced column(s) *`refcolumn`* of the referenced table *`reftable`*. If *`refcolumn`* is omitted, the primary key of the *`reftable`* is used. The referenced columns must be the columns of a unique or primary key constraint in the referenced table.

A value added to these columns is matched against the values of the referenced table and referenced columns using the given match type. There are three match types: `MATCH FULL`, `MATCH PARTIAL`, and a default match type if none is specified. `MATCH FULL` will not allow one column of a multicolumn foreign key to be NULL unless all foreign key columns are NULL. The default match type allows some foreign key columns to be NULL while other parts of the foreign key are not NULL. `MATCH PARTIAL` is not yet implemented.

In addition, when the data in the referenced columns is changed, certain actions are performed on the data in this table's columns. The `ON DELETE` clause specifies the action to do when a referenced row in the referenced table is being deleted. Likewise, the `ON UPDATE` clause specifies the action to perform when a referenced column in the referenced table is being updated to a new value. If the row is updated, but the referenced column is not actually changed, no action is done. There are the following possible actions for each clause:

`NO ACTION`

Produce an error indicating that the deletion or update would create a foreign key constraint violation. This is the default action.

`RESTRICT`

Same as `NO ACTION`.

`CASCADE`

Delete any rows referencing the deleted row, or update the value of the referencing column to the new value of the referenced column, respectively.

`SET NULL`

Set the referencing column values to NULL.

`SET DEFAULT`

Set the referencing column values to their default value.

If primary key column is updated frequently, it may be wise to add an index to the `REFERENCES` column so that `NO ACTION` and `CASCADE` actions associated with the `REFERENCES` column can be more efficiently performed.

`DEFERRABLE` or `NOT DEFERRABLE`

This controls whether the constraint can be deferred. A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable may be postponed until

the end of the transaction (using the SET CONSTRAINTS command). `NOT DEFERRABLE` is the default. Only foreign key constraints currently accept this clause. All other constraint types are not deferrable.

`INITIALLY IMMEDIATE` or `INITIALLY DEFERRED`

If a constraint is deferrable, this clause specifies the default time to check the constraint. If the constraint is `INITIALLY IMMEDIATE`, it is checked after each statement. This is the default. If the constraint is `INITIALLY DEFERRED`, it is checked only at the end of the transaction. The constraint check time can be altered with the SET CONSTRAINTS command.

Diagnostics

`CREATE`

Message returned if table is successfully created.

`ERROR`

Message returned if table creation failed. This is usually accompanied by some descriptive text, such as: `ERROR: Relation '`*`table`*`' already exists`, which occurs at runtime if the table specified already exists in the database.

Notes

- Whenever an application makes use of OIDs to identify specific rows of a table, it is recommended to create a unique constraint on the `oid` column of that table, to ensure that OIDs in the table will indeed uniquely identify rows even after counter wraparound. Avoid assuming that OIDs are unique across tables; if you need a database-wide unique identifier, use the combination of `tableoid` and row OID for the purpose. (It is likely that future PostgreSQL releases will use a separate OID counter for each table, so that it will be *necessary*, not optional, to include `tableoid` to have a unique identifier database-wide.)

  ## Tip

  The use of `WITHOUT OIDS` is not recommended for tables with no primary key, since without either an OID or a unique data key, it is difficult to identify specific rows.

- PostgreSQL automatically creates an index for each unique constraint and primary key constraint to enforce the uniqueness. Thus, it is not necessary to create an explicit index for primary key columns. (See CREATE INDEX for more information.)

- The SQL92 standard says that `CHECK` column constraints may only refer to the column they apply to; only `CHECK` table constraints may refer to multiple columns. PostgreSQL does not enforce this restriction; it treats column and table check constraints alike.

- Unique constraints and primary keys are not inherited in the current implementation. This makes the combination of inheritance and unique constraints rather dysfunctional.

Examples

Create table `films` and table `distributors`:

```
CREATE TABLE films (
```

```
    code        CHARACTER(5) CONSTRAINT firstkey PRIMARY KEY,
    title       CHARACTER VARYING(40) NOT NULL,
    did         DECIMAL(3) NOT NULL,
    date_prod   DATE,
    kind        CHAR(10),
    len         INTERVAL HOUR TO MINUTE
);

CREATE TABLE distributors (
     did    DECIMAL(3) PRIMARY KEY DEFAULT NEXTVAL('serial'),
     name   VARCHAR(40) NOT NULL CHECK (name <> '')
);
```

Create a table with a 2-dimensional array:

```
CREATE TABLE array (
    vector  INT[][]
);
```

Define a unique table constraint for the table films. Unique table constraints can be defined on one or more columns of the table:

```
CREATE TABLE films (
    code        CHAR(5),
    title       VARCHAR(40),
    did         DECIMAL(3),
    date_prod   DATE,
    kind        VARCHAR(10),
    len         INTERVAL HOUR TO MINUTE,
    CONSTRAINT production UNIQUE(date_prod)
);
```

Define a check column constraint:

```
CREATE TABLE distributors (
    did     DECIMAL(3) CHECK (did > 100),
    name    VARCHAR(40)
);
```

Define a check table constraint:

```
CREATE TABLE distributors (
    did     DECIMAL(3),
    name    VARCHAR(40)
    CONSTRAINT con1 CHECK (did > 100 AND name <> '')
);
```

Define a primary key table constraint for the table `films`. Primary key table constraints can be defined on one or more columns of the table.

```
CREATE TABLE films (
    code        CHAR(5),
    title       VARCHAR(40),
    did         DECIMAL(3),
    date_prod   DATE,
    kind        VARCHAR(10),
```

```
    len         INTERVAL HOUR TO MINUTE,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

Define a primary key constraint for table `distributors`. The following two examples are equivalent, the first using the table constraint syntax, the second the column constraint notation.

```
CREATE TABLE distributors (
    did     DECIMAL(3),
    name    CHAR VARYING(40),
    PRIMARY KEY(did)
);

CREATE TABLE distributors (
    did     DECIMAL(3) PRIMARY KEY,
    name    VARCHAR(40)
);
```

This assigns a literal constant default value for the column `name`, and arranges for the default value of column `did` to be generated by selecting the next value of a sequence object. The default value of `modtime` will be the time at which the row is inserted.

```
CREATE TABLE distributors (
    name      VARCHAR(40) DEFAULT 'luso films',
    did       INTEGER DEFAULT NEXTVAL('distributors_serial'),
    modtime   TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Define two `NOT NULL` column constraints on the table `distributors`, one of which is explicitly given a name:

```
CREATE TABLE distributors (
    did     DECIMAL(3) CONSTRAINT no_null NOT NULL,
    name    VARCHAR(40) NOT NULL
);
```

Define a unique constraint for the `name` column:

```
CREATE TABLE distributors (
    did     DECIMAL(3),
    name    VARCHAR(40) UNIQUE
);
```

The above is equivalent to the following specified as a table constraint:

```
CREATE TABLE distributors (
    did     DECIMAL(3),
    name    VARCHAR(40),
    UNIQUE(name)
);
```

Compatibility

The `CREATE TABLE` conforms to SQL92 Intermediate and to a subset of SQL99, with exceptions listed below and in the descriptions above.

Temporary Tables

In addition to the local temporary table, SQL92 also defines a `CREATE GLOBAL TEMPORARY TABLE` statement. Global temporary tables are also visible to other sessions.

For temporary tables, there is an optional `ON COMMIT` clause:

```
CREATE { GLOBAL | LOCAL } TEMPORARY TABLE table ( ... ) [ ON COMMIT
 { DELETE | PRESERVE } ROWS ]
```

The `ON COMMIT` clause specifies whether or not the temporary table should be emptied of rows whenever `COMMIT` is executed. If the `ON COMMIT` clause is omitted, SQL92 specifies that the default is `ON COMMIT DELETE ROWS`. However, the behavior of PostgreSQL is always like `ON COMMIT PRESERVE ROWS`.

`NULL` "Constraint"

The `NULL` "constraint" (actually a non-constraint) is a PostgreSQL extension to SQL92 that is included for compatibility with some other RDBMS (and for symmetry with the `NOT NULL` constraint). Since it is the default for any column, its presence is simply noise.

Assertions

An assertion is a special type of integrity constraint and shares the same namespace as other constraints. However, an assertion is not necessarily dependent on one particular table as constraints are, so SQL92 provides the `CREATE ASSERTION` statement as an alternate method for defining a constraint:

```
CREATE ASSERTION name CHECK ( condition )
```

PostgreSQL does not implement assertions at present.

Inheritance

Multiple inheritance via the `INHERITS` clause is a PostgreSQL language extension. SQL99 (but not SQL92) defines single inheritance using a different syntax and different semantics. SQL99-style inheritance is not yet supported by PostgreSQL.

Object IDs

The PostgreSQL concept of OIDs is not standard.

See Also

ALTER TABLE , DROP TABLE

Name

CREATE TABLE AS — create a new table from the results of a query

Synopsis

```
CREATE [ [ LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name
 [ (column_name [, ...] ) ]
    AS query
```

Description

`CREATE TABLE AS` creates a table and fills it with data computed by a `SELECT` command. The table columns have the names and data types associated with the output columns of the `SELECT` (except that you can override the column names by giving an explicit list of new column names).

`CREATE TABLE AS` bears some resemblance to creating a view, but it is really quite different: it creates a new table and evaluates the query just once to fill the new table initially. The new table will not track subsequent changes to the source tables of the query. In contrast, a view re-evaluates the underlying `SELECT` statements whenever it is queried.

Parameters

`[LOCAL] TEMPORARY` or `[LOCAL] TEMP`

If specified, the table is created as a temporary table. Temporary tables are automatically dropped at the end of a session. Existing persistent tables with the same name are not visible to the current session while the temporary table exists. Any indexes created on a temporary table are automatically temporary as well.

The `LOCAL` word is optional.

`table_name`

The name of the new table to be created. This table must not already exist. However, a temporary table can be created that has the same name as an existing permanent table.

`column_name`

The name of a column in the new table. Multiple column names can be specified using a comma-delimited list of column names. If column names are not provided, they are taken from the output column names of the query.

`query`

A query statement (that is, a `SELECT` command). Refer to SELECT for a description of the allowed syntax.

Diagnostics

Refer to CREATE TABLE and SELECT for a summary of possible output messages.

Notes

This command is functionally equivalent to SELECT INTO , but it is preferred since it is less likely to be confused with other uses of the `SELECT ... INTO` syntax.

Compatibility

This command is modeled after an Oracle feature. There is no command with equivalent functionality in SQL92 or SQL99. However, a combination of `CREATE TABLE` and `INSERT ... SELECT` can accomplish the same thing with little more effort.

History

The `CREATE TABLE AS` command has been available since PostgreSQL 6.3.

See Also

CREATE TABLE, CREATE VIEW , SELECT, SELECT INTO

<docinfo>2001-09-13</docinfo>

Name

CREATE TRIGGER — define a new trigger

Synopsis

<refsynopsisdivinfo>2000-03-25</refsynopsisdivinfo>

```
CREATE TRIGGER name { BEFORE | AFTER } { event [OR ...] }
    ON table FOR EACH { ROW | STATEMENT }
    EXECUTE PROCEDURE func ( arguments )
```

Inputs

name

The name to give the new trigger.

table

The name of an existing table.

event

One of INSERT, DELETE or UPDATE.

func

A user-supplied function.

Outputs

CREATE

This message is returned if the trigger is successfully created.

Description

CREATE TRIGGER will enter a new trigger into the current data base. The trigger will be associated with the relation *table* and will execute the specified function *func*.

The trigger can be specified to fire either before BEFORE the operation is attempted on a tuple (before constraints are checked and the INSERT, UPDATE or DELETE is attempted) or AFTER the operation has been attempted (e.g., after constraints are checked and the INSERT, UPDATE or DELETE has completed). If the trigger fires before the event, the trigger may skip the operation for the current tuple, or change the tuple being inserted (for INSERT and UPDATE operations only). If the trigger fires after the event, all changes, including the last insertion, update, or deletion, are "visible" to the trigger.

SELECT does not modify any rows so you can not create SELECT triggers. Rules and views are more appropriate in such cases.

Refer to the chapters on SPI and Triggers in the *PostgreSQL Programmer's Guide* for more information.

Notes

To create a trigger on a table, the user must have the TRIGGER privilege on the table.

As of the current release, STATEMENT triggers are not implemented.

Refer to the DROP TRIGGER command for information on how to remove triggers.

Examples

Check if the specified distributor code exists in the distributors table before appending or updating a row in the table films:

```
CREATE TRIGGER if_dist_exists
    BEFORE INSERT OR UPDATE ON films FOR EACH ROW
    EXECUTE PROCEDURE check_primary_key ('did', 'distributors',
 'did');
```

Before cancelling a distributor or updating its code, remove every reference to the table films:

```
CREATE TRIGGER if_film_exists
    BEFORE DELETE OR UPDATE ON distributors FOR EACH ROW
    EXECUTE PROCEDURE check_foreign_key (1, 'CASCADE', 'did', 'films',
 'did');
```

The second example can also be done by using a foreign key, constraint as in:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40),
    CONSTRAINT if_film_exists
    FOREIGN KEY(did) REFERENCES films
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

Compatibility

SQL92

There is no CREATE TRIGGER statement in SQL92.

SQL99

The CREATE TRIGGER statement in PostgreSQL implements a subset of the SQL99 standard. The following functionality is missing:

- SQL99 allows triggers to fire on updates to specific columns (e.g., AFTER UPDATE OF col1, col2).
- SQL99 allows you to define aliases for the “old” and “new” rows or tables for use in the definition of the triggered action (e.g., CREATE TRIGGER ... ON tablename REFERENCING OLD ROW AS somename NEW ROW AS othername ...). Since PostgreSQL allows trigger procedures to be written in any number of user-defined languages, access to the data is handled in a language-specific way.

- PostgreSQL only has row-level triggers, no statement-level triggers.
- PostgreSQL only allows the execution of a stored procedure for the triggered action. SQL99 allows the execution of a number of other SQL commands, such as `CREATE TABLE` as triggered action. This limitation is not hard to work around by creating a stored procedure that executes these commands.

See Also

CREATE FUNCTION, DROP TRIGGER , *PostgreSQL Programmer's Guide*

Name

CREATE TYPE — define a new data type

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
CREATE TYPE typename ( INPUT = input_function, OUTPUT
 = output_function
      , INTERNALLENGTH = { internallength | VARIABLE }
    [ , EXTERNALLENGTH = { externallength | VARIABLE } ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ] [ , DELIMITER = delimiter ]
    [ , SEND = send_function ] [ , RECEIVE = receive_function ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
)
```

Inputs

typename

The name of a type to be created.

internallength

A literal value, which specifies the internal length of the new type.

externallength

A literal value, which specifies the external (displayed) length of the new type.

input_function

The name of a function, created by `CREATE FUNCTION`, which converts data from its external form to the type's internal form.

output_function

The name of a function, created by `CREATE FUNCTION`, which converts data from its internal form to a form suitable for display.

element

The type being created is an array; this specifies the type of the array elements.

delimiter

The delimiter character to be used between values in arrays made of this type.

default

The default value for the data type. Usually this is omitted, so that the default is NULL.

send_function

The name of a function, created by CREATE FUNCTION, which converts data of this type into a form suitable for transmission to another machine.

receive_function

The name of a function, created by CREATE FUNCTION, which converts data of this type from a form suitable for transmission from another machine to internal form.

alignment

Storage alignment requirement of the data type. If specified, must be char, int2, int4, or double; the default is int4.

storage

Storage technique for the data type. If specified, must be plain, external, extended, or main; the default is plain.

Outputs

CREATE

Message returned if the type is successfully created.

Description

CREATE TYPE allows the user to register a new user data type with PostgreSQL for use in the current data base. The user who defines a type becomes its owner. *typename* is the name of the new type and must be unique within the types defined for this database.

CREATE TYPE requires the registration of two functions (using CREATE FUNCTION) before defining the type. The representation of a new base type is determined by *input_function*, which converts the type's external representation to an internal representation usable by the operators and functions defined for the type. Naturally, *output_function* performs the reverse transformation. The input function may be declared as taking one argument of type opaque, or as taking three arguments of types opaque, OID, int4. (The first argument is the input text as a C string, the second argument is the element type in case this is an array type, and the third is the typmod of the destination column, if known.) The output function may be declared as taking one argument of type opaque, or as taking two arguments of types opaque, OID. (The first argument is actually of the data type itself, but since the output function must be declared first, it's easier to declare it as accepting type opaque. The second argument is again the array element type for array types.)

New base data types can be fixed length, in which case *internallength* is a positive integer, or variable length, indicated by setting *internallength* to VARIABLE. (Internally, this is represented by setting typlen to -1.) The internal representation of all variable-length types must start with an integer giving the total length of this value of the type.

The external representation length is similarly specified using the *externallength* keyword. (This value is not presently used, and is typically omitted, letting it default to VARIABLE.)

To indicate that a type is an array, specify the type of the array elements using the ELEMENT keyword. For example, to define an array of 4-byte integers ("int4"), specify

```
ELEMENT = int4
```

More details about array types appear below.

To indicate the delimiter to be used between values in the external representation of arrays of this type, *delimiter* can be set to a specific character. The default delimiter is the comma (','). Note that the delimiter is associated with the array element type, not the array type itself.

A default value may be specified, in case a user wants columns of the data type to default to something other than NULL. Specify the default with the DEFAULT keyword. (Such a default may be overridden by an explicit DEFAULT clause attached to a particular column.)

The optional arguments *send_function* and *receive_function* are not currently used, and are usually omitted (allowing them to default to the *output_function* and *input_function* respectively). These functions may someday be resurrected for use in specifying machine-independent binary representations.

The optional flag, PASSEDBYVALUE, indicates that values of this data type are passed by value rather than by reference. Note that you may not pass by value types whose internal representation is longer than the width of the Datum type (four bytes on most machines, eight bytes on a few).

The *alignment* keyword specifies the storage alignment required for the data type. The allowed values equate to alignment on 1, 2, 4, or 8 byte boundaries. Note that variable-length types must have an alignment of at least 4, since they necessarily contain an int4 as their first component.

The *storage* keyword allows selection of storage strategies for variable-length data types (only plain is allowed for fixed-length types). plain disables TOAST for the data type: it will always be stored in-line and not compressed. extended gives full TOAST capability: the system will first try to compress a long data value, and will move the value out of the main table row if it's still too long. external allows the value to be moved out of the main table, but the system will not try to compress it. main allows compression, but discourages moving the value out of the main table. (Data items with this storage method may still be moved out of the main table if there is no other way to make a row fit, but they will be kept in the main table preferentially over extended and external items.)

Array Types

Whenever a user-defined data type is created, PostgreSQL automatically creates an associated array type, whose name consists of the base type's name prepended with an underscore. The parser understands this naming convention, and translates requests for columns of type foo[] into requests for type _foo. The implicitly-created array type is variable length and uses the built-in input and output functions array_in and array_out.

You might reasonably ask "why is there an ELEMENT option, if the system makes the correct array type automatically?" The only case where it's useful to use ELEMENT is when you are making a fixed-length type that happens to be internally an array of N identical things, and you want to allow the N things to be accessed directly by subscripting, in addition to whatever operations you plan to provide for the type as a whole. For example, type name allows its constituent chars to be accessed this way. A 2-D point type could allow its two component floats to be accessed like point[0] and point[1]. Note that this facility only works for fixed-length types whose internal form is exactly a sequence of N identical fields. A subscriptable variable-length type must have the generalized internal representation used by array_in and array_out. For historical reasons (i.e., this is clearly wrong but it's far too late to change it), subscripting of fixed-length array types starts from zero, rather than from one as for variable-length arrays.

Notes

User-defined type names cannot begin with the underscore character ("_") and can only be 30 characters long (or in general NAMEDATALEN-2, rather than the NAMEDATALEN-1 characters allowed for other names). Type names beginning with underscore are reserved for internally-created array type names.

Examples

This example creates the `box` data type and then uses the type in a table definition:

```
CREATE TYPE box (INTERNALLENGTH = 16,
    INPUT = my_procedure_1, OUTPUT = my_procedure_2);
CREATE TABLE myboxes (id INT4, description box);
```

If `box`'s internal structure were an array of four `float4`s, we might instead say

```
CREATE TYPE box (INTERNALLENGTH = 16,
    INPUT = my_procedure_1, OUTPUT = my_procedure_2,
    ELEMENT = float4);
```

which would allow a box value's component floats to be accessed by subscripting. Otherwise the type behaves the same as before.

This example creates a large object type and uses it in a table definition:

```
CREATE TYPE bigobj (INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE);
CREATE TABLE big_objs (id int4, obj bigobj);
```

Compatibility

This `CREATE TYPE` command is a PostgreSQL extension. There is a `CREATE TYPE` statement in SQL99 that is rather different in detail.

See Also

CREATE FUNCTION, DROP TYPE , *PostgreSQL Programmer's Guide*

Name

CREATE USER — define a new database user account

Synopsis

<refsynopsisdivinfo>2001-07-10</refsynopsisdivinfo>

```
CREATE USER username [ [ WITH ] option [ ... ] ]

where option can be:

    SYSID uid
        | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
        | CREATEDB | NOCREATEDB
        | CREATEUSER | NOCREATEUSER
        | IN GROUP groupname [, ...]
        | VALID UNTIL 'abstime'
```

Inputs

username

The name of the user.

uid

The SYSID clause can be used to choose the PostgreSQL user id of the user that is being created. It is not at all necessary that those match the UNIX user ids, but some people choose to keep the numbers the same.

If this is not specified, the highest assigned user id plus one (with a minimum of 100) will be used as default.

password

Sets the user's password. If you do not plan to use password authentication you can omit this option, but the user won't be able to connect to a password-authenticated server. The password can be set or changed later, using ALTER USER .

ENCRYPTED
UNENCRYPTED

These keywords control whether the password is stored encrypted in pg_shadow. (If neither is specified, the default behavior is determined by the PASSWORD_ENCRYPTION server parameter.) If the presented string is already in MD5-encrypted format, then it is stored as-is, regardless of whether ENCRYPTED or UNENCRYPTED is specified. This allows reloading of encrypted passwords during dump/restore.

See the chapter on client authentication in the *Administrator's Guide* for details on how to set up authentication mechanisms. Note that older clients may lack support for the MD5 authentication mechanism that's needed to work with passwords that are stored encrypted.

CREATEDB
NOCREATEDB

These clauses define a user's ability to create databases. If CREATEDB is specified, the user being defined will be allowed to create his own databases. Using NOCREATEDB will deny a user the ability to create databases. If this clause is omitted, NOCREATEDB is used by default.

CREATEUSER
NOCREATEUSER

These clauses determine whether a user will be permitted to create new users himself. This option will also make the user a superuser who can override all access restrictions. Omitting this clause will set the user's value of this attribute to be NOCREATEUSER.

groupname

A name of a group into which to insert the user as a new member. Multiple group names may be listed.

abstime

The VALID UNTIL clause sets an absolute time after which the user's password is no longer valid. If this clause is omitted the login will be valid for all time.

Outputs

`CREATE USER`

Message returned if the command completes successfully.

Description

`CREATE USER` will add a new user to an instance of PostgreSQL. Refer to the administrator's guide for information about managing users and authentication. You must be a database superuser to use this command.

Use ALTER USER to change a user's password and privileges, and DROP USER to remove a user. Use ALTER GROUP to add or remove the user from other groups. PostgreSQL comes with a script createuser which has the same functionality as this command (in fact, it calls this command) but can be run from the command shell.

Usage

Create a user with no password:

```
CREATE USER jonathan
```

Create a user with a password:

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

Create a user with a password, whose account is valid until the end of 2001. Note that after one second has ticked in 2002, the account is not valid:

```
CREATE USER miriam WITH PASSWORD 'jw8s0F4' VALID UNTIL 'Jan 1 2002';
```

Create an account where the user can create databases:

```
CREATE USER manuel WITH PASSWORD 'jw8s0F4' CREATEDB;
```

Compatibility

SQL92

There is no `CREATE USER` statement in SQL92.

Name

CREATE VIEW — define a new view

Synopsis

<refsynopsisdivinfo>2000-03-25</refsynopsisdivinfo>

```
CREATE VIEW view [ ( column name list ) ] AS SELECT query
```

Inputs

view

The name of a view to be created.

column name list

An optional list of names to be used for columns of the view. If given, these names override the column names that would be deduced from the SQL query.

query

An SQL query which will provide the columns and rows of the view.

Refer to the SELECT statement for more information about valid arguments.

Outputs

CREATE

The message returned if the view is successfully created.

ERROR: Relation '*view*' already exists

This error occurs if the view specified already exists in the database.

NOTICE: Attribute '*column*' has an unknown type

The view will be created having a column with an unknown type if you do not specify it. For example, the following command gives a warning:

```
CREATE VIEW vista AS SELECT 'Hello World'
```

whereas this command does not:

```
CREATE VIEW vista AS SELECT text 'Hello World'
```

Description

CREATE VIEW will define a view of a table. The view is not physically materialized. Instead, a query rewrite retrieve rule is automatically generated to support retrieve operations on views.

Notes

Currently, views are read only: the system will not allow an insert, update, or delete on a view. You can get the effect of an updatable view by creating rules that rewrite inserts, etc. on the view into appropriate actions on other tables. For more information see CREATE RULE.

Use the `DROP VIEW` statement to drop views.

Usage

Create a view consisting of all Comedy films:

```
CREATE VIEW kinds AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';

SELECT * FROM kinds;

 code  |           title           | did | date_prod  |  kind  | len
-------+---------------------------+-----+------------+--------
+-------
 UA502 | Bananas                   | 105 | 1971-07-13 | Comedy | 01:22
 C_701 | There's a Girl in my Soup | 107 | 1970-06-11 | Comedy | 01:36
(2 rows)
```

Compatibility

SQL92

SQL92 specifies some additional capabilities for the `CREATE VIEW` statement:

```
CREATE VIEW view [ column [, ...] ]
    AS SELECT expression [ AS colname ] [, ...]
    FROM table [ WHERE condition ]
    [ WITH [ CASCADE | LOCAL ] CHECK OPTION ]
```

The optional clauses for the full SQL92 command are:

CHECK OPTION

This option is to do with updatable views. All INSERTs and UPDATEs on the view will be checked to ensure data satisfy the view-defining condition. If they do not, the update will be rejected.

LOCAL

Check for integrity on this view.

CASCADE

Check for integrity on this view and on any dependent view. CASCADE is assumed if neither CASCADE nor LOCAL is specified.

Name

DECLARE — define a cursor

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DECLARE cursorname [ BINARY ] [ INSENSITIVE ] [ SCROLL ]
    CURSOR FOR query
    [ FOR { READ ONLY | UPDATE [ OF column [, ...] ] } ]
```

Inputs

cursorname

The name of the cursor to be used in subsequent FETCH operations.

BINARY

Causes the cursor to fetch data in binary rather than in text format.

INSENSITIVE

SQL92 keyword indicating that data retrieved from the cursor should be unaffected by updates from other processes or cursors. Since cursor operations occur within transactions in PostgreSQL this is always the case. This keyword has no effect.

SCROLL

SQL92 keyword indicating that data may be retrieved in multiple rows per FETCH operation. Since this is allowed at all times by PostgreSQL this keyword has no effect.

query

An SQL query which will provide the rows to be governed by the cursor. Refer to the SELECT statement for further information about valid arguments.

READ ONLY

SQL92 keyword indicating that the cursor will be used in a read only mode. Since this is the only cursor access mode available in PostgreSQL this keyword has no effect.

UPDATE

SQL92 keyword indicating that the cursor will be used to update tables. Since cursor updates are not currently supported in PostgreSQL this keyword provokes an informational error message.

column

Column(s) to be updated. Since cursor updates are not currently supported in PostgreSQL the UPDATE clause provokes an informational error message.

Outputs

SELECT

The message returned if the SELECT is run successfully.

`NOTICE: Closing pre-existing portal "`*`cursorname`*`"`

This message is reported if the same cursor name was already declared in the current transaction block. The previous definition is discarded.

`ERROR: DECLARE CURSOR may only be used in begin/end transaction blocks`

This error occurs if the cursor is not declared within a transaction block.

Description

`DECLARE` allows a user to create cursors, which can be used to retrieve a small number of rows at a time out of a larger query. Cursors can return data either in text or in binary format using FETCH .

Normal cursors return data in text format, either ASCII or another encoding scheme depending on how the PostgreSQL backend was built. Since data is stored natively in binary format, the system must do a conversion to produce the text format. In addition, text formats are often larger in size than the corresponding binary format. Once the information comes back in text form, the client application may need to convert it to a binary format to manipulate it. BINARY cursors give you back the data in the native binary representation.

As an example, if a query returns a value of one from an integer column, you would get a string of `1` with a default cursor whereas with a binary cursor you would get a 4-byte value equal to control-A (`^A`).

BINARY cursors should be used carefully. User applications such as psql are not aware of binary cursors and expect data to come back in a text format.

String representation is architecture-neutral whereas binary representation can differ between different machine architectures. *PostgreSQL does not resolve byte ordering or representation issues for binary cursors*. Therefore, if your client machine and server machine use different representations (e.g., “big-endian” versus “little-endian”), you will probably not want your data returned in binary format. However, binary cursors may be a little more efficient since there is less conversion overhead in the server to client data transfer.

Tip

If you intend to display the data in ASCII, getting it back in ASCII will save you some effort on the client side.

Notes

Cursors are only available in transactions. Use to BEGIN , COMMIT and ROLLBACK to define a transaction block.

In SQL92 cursors are only available in embedded SQL (ESQL) applications. The PostgreSQL backend does not implement an explicit `OPEN cursor` statement; a cursor is considered to be open when it is declared. However, ecpg, the embedded SQL preprocessor for PostgreSQL, supports the SQL92 cursor conventions, including those involving DECLARE and OPEN statements.

Usage

To declare a cursor:

```
DECLARE liahona CURSOR
    FOR SELECT * FROM films;
```

Compatibility

SQL92

SQL92 allows cursors only in embedded SQL and in modules. PostgreSQL permits cursors to be used interactively. SQL92 allows embedded or modular cursors to update database information. All PostgreSQL cursors are read only. The BINARY keyword is a PostgreSQL extension.

Name

DELETE — delete rows of a table

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DELETE FROM [ ONLY ] table [ WHERE condition ]
```

Inputs

table

The name of an existing table.

condition

This is an SQL selection query which returns the rows which are to be deleted.

Refer to the SELECT statement for further description of the WHERE clause.

Outputs

DELETE *count*

Message returned if items are successfully deleted. The *count* is the number of rows deleted.

If *count* is 0, no rows were deleted.

Description

DELETE removes rows which satisfy the WHERE clause from the specified table.

If the *condition* (WHERE clause) is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.

Tip

TRUNCATE is a PostgreSQL extension which provides a faster mechanism to remove all rows from a table.

By default DELETE will delete tuples in the table specified and all its sub-tables. If you wish to only update the specific table mentioned, you should use the ONLY clause.

You must have write access to the table in order to modify it, as well as read access to any table whose values are read in the *condition*.

Usage

Remove all films but musicals:

```
DELETE FROM films WHERE kind <> 'Musical';
SELECT * FROM films;
```

```
 code  |           title           | did | date_prod  |  kind   | len
-------+---------------------------+-----+------------+---------
+-------
 UA501 | West Side Story           | 105 | 1961-01-03 | Musical |
 02:32
 TC901 | The King and I            | 109 | 1956-08-11 | Musical |
 02:13
 WD101 | Bed Knobs and Broomsticks | 111 |            | Musical |
 01:57
(3 rows)
```

Clear the table `films`:

```
DELETE FROM films;
SELECT * FROM films;

 code | title | did | date_prod | kind | len
------+-------+-----+-----------+------+-----
(0 rows)
```

Compatibility

SQL92

SQL92 allows a positioned DELETE statement:

```
DELETE FROM table WHERE
    CURRENT OF cursor
```

where *`cursor`* identifies an open cursor. Interactive cursors in PostgreSQL are read-only.

Name

DROP AGGREGATE — remove a user-defined aggregate function

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP AGGREGATE name ( type )
```

Inputs

name

The name of an existing aggregate function.

type

The input data type of an existing aggregate function, or * if the function accepts any input type. (Refer to the *PostgreSQL User's Guide* for further information about data types.)

Outputs

DROP

Message returned if the command is successful.

ERROR: RemoveAggregate: aggregate '*name*' for type *type* does not exist

This message occurs if the aggregate function specified does not exist in the database.

Description

DROP AGGREGATE will remove all references to an existing aggregate definition. To execute this command the current user must be the owner of the aggregate.

Notes

Use CREATE AGGREGATE to create aggregate functions.

Usage

To remove the myavg aggregate for type int4:

```
DROP AGGREGATE myavg(int4);
```

Compatibility

SQL92

There is no DROP AGGREGATE statement in SQL92; the statement is a PostgreSQL language extension.

Name

DROP DATABASE — remove a database

Synopsis

<refsynopsisdivinfo>1999-12-11</refsynopsisdivinfo>

```
DROP DATABASE name
```

Inputs

name

The name of an existing database to remove.

Outputs

`DROP DATABASE`

This message is returned if the command is successful.

`DROP DATABASE: cannot be executed on the currently open database`

You cannot be connected to the database you are about to remove. Instead, connect to `template1` or any other database and run this command again.

`DROP DATABASE: may not be called in a transaction block`

You must finish the transaction in progress before you can call this command.

Description

`DROP DATABASE` removes the catalog entries for an existing database and deletes the directory containing the data. It can only be executed by the database owner (usually the user that created it).

`DROP DATABASE` cannot be undone. Use it with care!

Notes

This command cannot be executed while connected to the target database. Thus, it might be more convenient to use the shell script dropdb, which is a wrapper around this command, instead.

Refer to CREATE DATABASE for information on how to create a database.

Compatibility

SQL92

`DROP DATABASE` statement is a PostgreSQL language extension; there is no such command in SQL92.

Name

DROP FUNCTION — remove a user-defined function

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP FUNCTION name ( [ type [, ...] ] )
```

Inputs

name

The name of an existing function.

type

The type of the function's parameters.

Outputs

DROP

Message returned if the command completes successfully.

NOTICE RemoveFunction: Function "*name*" ("*types*") does not exist

This message is given if the function specified does not exist in the current database.

Description

DROP FUNCTION will remove the definition of an existing function. To execute this command the user must be the owner of the function. The input argument types to the function must be specified, since several different functions may exist with the same name and different argument lists.

Notes

Refer to CREATE FUNCTION for information on creating functions.

No checks are made to ensure that types, operators, access methods, or triggers that rely on the function have been removed first.

Examples

This command removes the square root function:

```
DROP FUNCTION sqrt(integer);
```

Compatibility

A DROP FUNCTION statement is defined in SQL99. One of its syntax forms is:

```
DROP FUNCTION name (arg, ...) { RESTRICT | CASCADE }
```

where `CASCADE` specifies dropping all objects that depend on the function and `RESTRICT` refuses to drop the function if dependent objects exist.

See Also

CREATE FUNCTION

Name

DROP GROUP — remove a user group

Synopsis

<refsynopsisdivinfo>2000-01-14</refsynopsisdivinfo>

```
DROP GROUP name
```

Inputs

name

The name of an existing group.

Outputs

`DROP GROUP`

The message returned if the group is successfully deleted.

Description

`DROP GROUP` removes the specified group from the database. The users in the group are not deleted.

Use CREATE GROUP to add new groups, and ALTER GROUP to change a group's membership.

Usage

To drop a group:

```
DROP GROUP staff;
```

Compatibility

SQL92

There is no `DROP GROUP` in SQL92.

Name

DROP INDEX — remove an index

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP INDEX index_name [, ...]
```

Inputs

index_name

The name of an index to remove.

Outputs

`DROP`

The message returned if the command completes successfully.

`ERROR: index "index_name" does not exist`

This message occurs if *index_name* is not an index in the database.

Description

`DROP INDEX` drops an existing index from the database system. To execute this command you must be the owner of the index.

Notes

`DROP INDEX` is a PostgreSQL language extension.

Refer to CREATE INDEX for information on how to create indexes.

Usage

This command will remove the `title_idx` index:

```
DROP INDEX title_idx;
```

Compatibility

SQL92

SQL92 defines commands by which to access a generic relational database. Indexes are an implementation-dependent feature and hence there are no index-specific commands or definitions in the SQL92 language.

Name

DROP LANGUAGE — remove a user-defined procedural language

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP [ PROCEDURAL ] LANGUAGE name
```

Inputs

name

The name of an existing procedural language. For backward compatibility, the name may be enclosed by single quotes.

Outputs

`DROP`

This message is returned if the language is successfully dropped.

`ERROR: Language "name" doesn't exist`

This message occurs if a language called *name* is not found in the database.

Description

`DROP PROCEDURAL LANGUAGE` will remove the definition of the previously registered procedural language called *name*.

Notes

The `DROP PROCEDURAL LANGUAGE` statement is a PostgreSQL language extension.

Refer to CREATE LANGUAGE for information on how to create procedural languages.

No checks are made if functions or trigger procedures registered in this language still exist. To re-enable them without having to drop and recreate all the functions, the pg_proc's prolang attribute of the functions must be adjusted to the new object ID of the recreated pg_language entry for the PL.

Usage

This command removes the PL/Sample language:

```
DROP LANGUAGE plsample;
```

Compatibility

SQL92

There is no `DROP PROCEDURAL LANGUAGE` in SQL92.

Name

DROP OPERATOR — remove a user-defined operator

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP OPERATOR id ( lefttype | NONE , righttype | NONE )
```

Inputs

id

The identifier of an existing operator.

lefttype

The type of the operator's left argument; write NONE if the operator has no left argument.

righttype

The type of the operator's right argument; write NONE if the operator has no right argument.

Outputs

DROP

The message returned if the command is successful.

ERROR: RemoveOperator: binary operator '*oper*' taking '*lefttype*' and '*righttype*' does not exist

This message occurs if the specified binary operator does not exist.

ERROR: RemoveOperator: left unary operator '*oper*' taking '*lefttype*' does not exist

This message occurs if the left unary operator specified does not exist.

ERROR: RemoveOperator: right unary operator '*oper*' taking '*righttype*' does not exist

This message occurs if the right unary operator specified does not exist.

Description

DROP OPERATOR drops an existing operator from the database. To execute this command you must be the owner of the operator.

The left or right type of a left or right unary operator, respectively, must be specified as NONE.

Notes

The DROP OPERATOR statement is a PostgreSQL language extension.

Refer to CREATE OPERATOR for information on how to create operators.

It is the user's responsibility to remove any access methods and operator classes that rely on the deleted operator.

Usage

Remove power operator `a^n` for `int4`:

```
DROP OPERATOR ^ (int4, int4);
```

Remove left unary negation operator (`! b`) for `boolean`:

```
DROP OPERATOR ! (none, bool);
```

Remove right unary factorial operator (`i !`) for `int4`:

```
DROP OPERATOR ! (int4, none);
```

Compatibility

SQL92

There is no `DROP OPERATOR` in SQL92.

Name

DROP RULE — remove a rewrite rule

Synopsis

<refsynopsisdivinfo>1998-09-22</refsynopsisdivinfo>

```
DROP RULE name [, ...]
```

Inputs

name

The name of an existing rule to drop.

Outputs

`DROP`

Message returned if successful.

`ERROR: Rule or view "name" not found`

This message occurs if the specified rule does not exist.

Description

`DROP RULE` drops a rule from the specified PostgreSQL rule system. PostgreSQL will immediately cease enforcing it and will purge its definition from the system catalogs.

Notes

The `DROP RULE` statement is a PostgreSQL language extension.

Refer to `CREATE RULE` for information on how to create rules.

Once a rule is dropped, access to historical information the rule has written may disappear.

Usage

To drop the rewrite rule `newrule`:

```
DROP RULE newrule;
```

Compatibility

SQL92

There is no `DROP RULE` in SQL92.

Name

DROP SEQUENCE — remove a sequence

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP SEQUENCE name [, ...]
```

Inputs

name

The name of a sequence.

Outputs

`DROP`

The message returned if the sequence is successfully dropped.

`ERROR: sequence "name" does not exist`

This message occurs if the specified sequence does not exist.

Description

`DROP SEQUENCE` removes sequence number generators from the data base. With the current implementation of sequences as special tables it works just like the `DROP TABLE` statement.

Notes

The `DROP SEQUENCE` statement is a PostgreSQL language extension.

Refer to the `CREATE SEQUENCE` statement for information on how to create a sequence.

Usage

To remove sequence `serial` from database:

```
DROP SEQUENCE serial;
```

Compatibility

SQL92

There is no `DROP SEQUENCE` in SQL92.

Name

DROP TABLE — remove a table

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP TABLE name [, ...]
```

Inputs

name

The name of an existing table to drop.

Outputs

`DROP`

The message returned if the command completes successfully.

`ERROR: table "name" does not exist`

If the specified table does not exist in the database.

Description

`DROP TABLE` removes tables from the database. Only its owner may destroy a table. A table may be emptied of rows, but not destroyed, by using `DELETE`.

If a table being destroyed has secondary indexes on it, they will be removed first. The removal of just a secondary index will not affect the contents of the underlying table.

Notes

Refer to `CREATE TABLE` and `ALTER TABLE` for information on how to create or modify tables.

Usage

To destroy two tables, `films` and `distributors`:

```
DROP TABLE films, distributors;
```

Compatibility

SQL92

SQL92 specifies some additional capabilities for DROP TABLE:

```
DROP TABLE table { RESTRICT | CASCADE }
```

RESTRICT

Ensures that only a table with no dependent views or integrity constraints can be destroyed.

CASCADE

Any referencing views or integrity constraints will also be dropped.

Tip

At present, to remove a referenced view you must drop it explicitly.

<docinfo>2001-09-13</docinfo>

Name

DROP TRIGGER — remove a trigger

Synopsis

<refsynopsisdivinfo>1998-09-22</refsynopsisdivinfo>

```
DROP TRIGGER name ON table
```

Inputs

name

The name of an existing trigger.

table

The name of a table.

Outputs

`DROP`

The message returned if the trigger is successfully dropped.

`ERROR: DropTrigger: there is no trigger name on relation "table"`

This message occurs if the trigger specified does not exist.

Description

`DROP TRIGGER` will remove all references to an existing trigger definition. To execute this command the current user must be the owner of the table for which the trigger is defined.

Examples

Destroy the `if_dist_exists` trigger on table `films`:

```
DROP TRIGGER if_dist_exists ON films;
```

Compatibility

SQL92

There is no `DROP TRIGGER` statement in SQL92.

SQL99

The `DROP TRIGGER` statement in PostgreSQL is incompatible with SQL99. In SQL99, trigger names are not local to tables, so the command is simply `DROP TRIGGER name`.

See Also

CREATE TRIGGER

Name

DROP TYPE — remove a user-defined data type

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP TYPE typename [, ...]
```

Inputs

typename

The name of an existing type.

Outputs

`DROP`

The message returned if the command is successful.

`ERROR: RemoveType: type '`*typename*`' does not exist`

This message occurs if the specified type is not found.

Description

`DROP TYPE` will remove a user type from the system catalogs.

Only the owner of a type can remove it.

Notes

- It is the user's responsibility to remove any operators, functions, aggregates, access methods, subtypes, and tables that use a deleted type. However, the associated array data type (which was automatically created by `CREATE TYPE`) will be removed automatically.
- If a built-in type is removed, the behavior of the server is unpredictable.

Examples

To remove the `box` type:

```
DROP TYPE box;
```

Compatibility

A `DROP TYPE` statement exists in SQL99. As with most other "drop" commands, `DROP TYPE` in SQL99 requires a "drop behavior" clause to select between dropping all dependent objects or refusing to drop if dependent objects exist:

```
DROP TYPE name { CASCADE | RESTRICT }
```

PostgreSQL currently ignores dependencies altogether.

Note that the `CREATE TYPE` command and the data type extension mechanisms in PostgreSQL differ from SQL99.

See Also

CREATE TYPE

Name

DROP USER — remove a database user account

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP USER name
```

Inputs

name

The name of an existing user.

Outputs

`DROP USER`

The message returned if the user is successfully deleted.

`ERROR: DROP USER: user "name" does not exist`

This message occurs if the user name is not found.

`DROP USER: user "name" owns database "name", cannot be removed`

You must drop the database first or change its ownership.

Description

`DROP USER` removes the specified user from the database. It does not remove tables, views, or other objects owned by the user. If the user owns any database you get an error.

Use CREATE USER to add new users, and ALTER USER to change a user's properties. PostgreSQL comes with a script dropuser which has the same functionality as this command (in fact, it calls this command) but can be run from the command shell.

Usage

To drop a user account:

```
DROP USER jonathan;
```

Compatibility

SQL92

There is no `DROP USER` in SQL92.

Name

DROP VIEW — remove a view

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP VIEW name [, ...]
```

Inputs

name

The name of an existing view.

Outputs

`DROP`

The message returned if the command is successful.

`ERROR: view name does not exist`

This message occurs if the specified view does not exist in the database.

Description

`DROP VIEW` drops an existing view from the database. To execute this command you must be the owner of the view.

Notes

Refer to `CREATE VIEW` for information on how to create views.

Usage

This command will remove the view called `kinds`:

```
DROP VIEW kinds;
```

Compatibility

SQL92

SQL92 specifies some additional capabilities for `DROP VIEW`:

```
DROP VIEW view { RESTRICT | CASCADE }
```

Inputs

RESTRICT

Ensures that only a view with no dependent views or integrity constraints can be destroyed.

CASCADE

Any referencing views and integrity constraints will be dropped as well.

Notes

At present, to remove a referenced view from a PostgreSQL database, you must drop it explicitly.

Name

END — commit the current transaction

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
END [ WORK | TRANSACTION ]
```

Inputs

WORK
TRANSACTION

Optional keywords. They have no effect.

Outputs

`COMMIT`

Message returned if the transaction is successfully committed.

```
NOTICE: COMMIT: no transaction in progress
```

If there is no transaction in progress.

Description

`END` is a PostgreSQL extension, and is a synonym for the SQL92-compatible COMMIT .

Notes

The keywords WORK and TRANSACTION are noise and can be omitted.

Use ROLLBACK to abort a transaction.

Usage

To make all changes permanent:

```
END WORK;
```

Compatibility

SQL92

`END` is a PostgreSQL extension which provides functionality equivalent to COMMIT .

Name

EXPLAIN — show the execution plan of a statement

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] query
```

Inputs

ANALYZE

Flag to carry out the query and show actual runtimes.

VERBOSE

Flag to show detailed query plan.

query

Any *query*.

Outputs

`NOTICE: QUERY PLAN:` *plan*

Explicit query plan from the PostgreSQL backend.

`EXPLAIN`

Flag sent after query plan is shown.

Description

This command displays the execution plan that the PostgreSQL planner generates for the supplied query. The execution plan shows how the table(s) referenced by the query will be scanned---by plain sequential scan, index scan, etc.---and if multiple tables are referenced, what join algorithms will be used to bring together the required tuples from each input table.

The most critical part of the display is the estimated query execution cost, which is the planner's guess at how long it will take to run the query (measured in units of disk page fetches). Actually two numbers are shown: the start-up time before the first tuple can be returned, and the total time to return all the tuples. For most queries the total time is what matters, but in contexts such as an EXISTS sub-query the planner will choose the smallest start-up time instead of the smallest total time (since the executor will stop after getting one tuple, anyway). Also, if you limit the number of tuples to return with a LIMIT clause, the planner makes an appropriate interpolation between the endpoint costs to estimate which plan is really the cheapest.

The ANALYZE option causes the query to be actually executed, not only planned. The total elapsed time expended within each plan node (in milliseconds) and total number of rows it actually returned are added to the display. This is useful for seeing whether the planner's estimates are close to reality.

The VERBOSE option emits the full internal representation of the plan tree, rather than just a summary (and sends it to the postmaster log file, too). Usually this option is only useful for debugging PostgreSQL.

Caution

Keep in mind that the query is actually executed when ANALYZE is used. Although `EXPLAIN` will discard any output that a SELECT would return, other side-effects of the query will happen as usual. If you wish to use `EXPLAIN ANALYZE` on an INSERT, UPDATE, or DELETE query without letting the query affect your data, use this approach:

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

Notes

There is only sparse documentation on the optimizer's use of cost information in PostgreSQL. Refer to the *User's Guide* and *Programmer's Guide* for more information.

Usage

To show a query plan for a simple query on a table with a single `int4` column and 128 rows:

```
EXPLAIN SELECT * FROM foo;
    NOTICE:  QUERY PLAN:

Seq Scan on foo  (cost=0.00..2.28 rows=128 width=4)

EXPLAIN
```

For the same table with an index to support an *equijoin* condition on the query, `EXPLAIN` will show a different plan:

```
EXPLAIN SELECT * FROM foo WHERE i = 4;
    NOTICE:  QUERY PLAN:

Index Scan using fi on foo  (cost=0.00..0.42 rows=1 width=4)

EXPLAIN
```

And finally, for the same table with an index to support an *equijoin* condition on the query, `EXPLAIN` will show the following for a query using an aggregate function:

```
EXPLAIN SELECT sum(i) FROM foo WHERE i = 4;
    NOTICE:  QUERY PLAN:

Aggregate  (cost=0.42..0.42 rows=1 width=4)
  ->  Index Scan using fi on foo  (cost=0.00..0.42 rows=1 width=4)
```

Note that the specific numbers shown, and even the selected query strategy, may vary between PostgreSQL releases due to planner improvements.

Compatibility

SQL92

There is no `EXPLAIN` statement defined in SQL92.

Name

FETCH — retrieve rows from a table using a cursor

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
FETCH [ direction ] [ count ] { IN | FROM } cursor
FETCH [ FORWARD | BACKWARD | RELATIVE ] [ # | ALL | NEXT | PRIOR ]
 { IN | FROM } cursor
```

Inputs

direction

selector defines the fetch direction. It can be one of the following:

FORWARD

fetch next row(s). This is the default if *selector* is omitted.

BACKWARD

fetch previous row(s).

RELATIVE

Noise word for SQL92 compatibility.

count

count determines how many rows to fetch. It can be one of the following:

#

A signed integer that specifies how many rows to fetch. Note that a negative integer is equivalent to changing the sense of FORWARD and BACKWARD.

ALL

Retrieve all remaining rows.

NEXT

Equivalent to specifying a count of `1`.

PRIOR

Equivalent to specifying a count of `-1`.

cursor

An open cursor's name.

Outputs

`FETCH` returns the results of the query defined by the specified cursor. The following messages will be returned if the query fails:

```
NOTICE: PerformPortalFetch: portal "cursor" not found
```

If *cursor* is not previously declared. The cursor must be declared within a transaction block.

```
NOTICE: FETCH/ABSOLUTE not supported, using RELATIVE
```

PostgreSQL does not support absolute positioning of cursors.

```
ERROR: FETCH/RELATIVE at current position is not supported
```

SQL92 allows one to repetitively retrieve the cursor at its "current position" using the syntax

```
FETCH RELATIVE 0 FROM cursor.
```

PostgreSQL does not currently support this notion; in fact the value zero is reserved to indicate that all rows should be retrieved and is equivalent to specifying the ALL keyword. If the RELATIVE keyword has been used, PostgreSQL assumes that the user intended SQL92 behavior and returns this error message.

Description

FETCH allows a user to retrieve rows using a cursor. The number of rows retrieved is specified by #. If the number of rows remaining in the cursor is less than #, then only those available are fetched. Substituting the keyword ALL in place of a number will cause all remaining rows in the cursor to be retrieved. Instances may be fetched in both FORWARD and BACKWARD directions. The default direction is FORWARD.

Tip

Negative numbers are allowed to be specified for the row count. A negative number is equivalent to reversing the sense of the FORWARD and BACKWARD keywords. For example, FORWARD -1 is the same as BACKWARD 1.

Notes

Note that the FORWARD and BACKWARD keywords are PostgreSQL extensions. The SQL92 syntax is also supported, specified in the second form of the command. See below for details on compatibility issues.

Updating data in a cursor is not supported by PostgreSQL, because mapping cursor updates back to base tables is not generally possible, as is also the case with VIEW updates. Consequently, users must issue explicit UPDATE commands to replace data.

Cursors may only be used inside of transactions because the data that they store spans multiple user queries.

Use MOVE to change cursor position. DECLARE will define a cursor. Refer to BEGIN , COMMIT , and ROLLBACK for further information about transactions.

Usage

The following examples traverses a table using a cursor.

```
-- Set up and use a cursor:

BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

```
-- Fetch first 5 rows in the cursor liahona:
FETCH FORWARD 5 IN liahona;

 code  |          title          | did | date_prod  |  kind    | len
-------+-------------------------+-----+------------+----------
+-------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    | 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantic | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action   | 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    | 02:28

-- Fetch previous row:
FETCH BACKWARD 1 IN liahona;

 code  | title   | did | date_prod  | kind   | len
-------+---------+-----+------------+--------+-------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08

-- close the cursor and commit work:

CLOSE liahona;
COMMIT WORK;
```

Compatibility

SQL92

Note

The non-embedded use of cursors is a PostgreSQL extension. The syntax and usage of cursors is being compared against the embedded form of cursors defined in SQL92.

SQL92 allows absolute positioning of the cursor for FETCH, and allows placing the results into explicit variables:

```
FETCH ABSOLUTE #
    FROM cursor
    INTO :variable [, ...]
```

ABSOLUTE

The cursor should be positioned to the specified absolute row number. All row numbers in PostgreSQL are relative numbers so this capability is not supported.

:variable

Target host variable(s).

Name

GRANT — define access privileges

Synopsis

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES |
 TRIGGER } [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] objectname [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...]
```

Description

The `GRANT` command gives specific permissions on an object (table, view, sequence) to one or more users or groups of users. These permissions are added to those already granted, if any.

The key word `PUBLIC` indicates that the privileges are to be granted to all users, including those that may be created later. `PUBLIC` may be thought of as an implicitly defined group that always includes all users. Note that any particular user will have the sum of privileges granted directly to him, privileges granted to any group he is presently a member of, and privileges granted to `PUBLIC`.

Users other than the creator of an object do not have any access privileges to the object unless the creator grants permissions. There is no need to grant privileges to the creator of an object, as the creator automatically holds all privileges. (The creator could, however, choose to revoke some of his own privileges for safety. Note that the ability to grant and revoke privileges is inherent in the creator and cannot be lost. The right to drop the object is likewise inherent in the creator, and cannot be granted or revoked.)

The possible privileges are:

SELECT

Allows SELECT from any column of the specified table, view, or sequence. Also allows the use of COPY FROM.

INSERT

Allows INSERT of a new row into the specified table. Also allows COPY TO.

UPDATE

Allows UPDATE of any column of the specified table. `SELECT ... FOR UPDATE` also requires this privilege (besides the `SELECT` privilege). For sequences, this privilege allows the use of `nextval`, `currval` and `setval`.

DELETE

Allows DELETE of a row from the specified table.

RULE

Allows the creation of a rule on the table/view. (See CREATE RULE statement.)

REFERENCES

To create a table with a foreign key constraint, it is necessary to have this privilege on the table with the referenced key.

TRIGGER

Allows the creation of a trigger on the specified table. (See CREATE TRIGGER statement.)

ALL PRIVILEGES

Grant all of the above privileges at once. The `PRIVILEGES` key word is optional in PostgreSQL, though it is required by strict SQL.

The privileges required by other commands are listed on the reference page of the respective command.

Notes

It should be noted that database *superusers* can access all objects regardless of object privilege settings. This is comparable to the rights of `root` in a Unix system. As with `root`, it's unwise to operate as a superuser except when absolutely necessary.

Currently, to grant privileges in PostgreSQL to only a few columns, you must create a view having the desired columns and then grant privileges to that view.

Use psql's `\z` command to obtain information about privileges on existing objects:

```
       Database    = lusitania
+------------------+-----------------------------------------------+
|  Relation        |        Grant/Revoke Permissions               |
+------------------+-----------------------------------------------+
| mytable          | {"=rw","miriam=arwdRxt","group todos=rw"}   |
+------------------+-----------------------------------------------+
Legend:
      uname=arwR -- privileges granted to a user
group gname=arwR -- privileges granted to a group
           =arwR -- privileges granted to PUBLIC

               r -- SELECT ("read")
               w -- UPDATE ("write")
               a -- INSERT ("append")
               d -- DELETE
               R -- RULE
               x -- REFERENCES
               t -- TRIGGER
         arwdRxt -- ALL PRIVILEGES
```

The REVOKE command is used to revoke access privileges.

Examples

Grant insert privilege to all users on table films:

```
GRANT INSERT ON films TO PUBLIC;
```

Grant all privileges to user `manuel` on view `kinds`:

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
```

Compatibility

SQL92

The `PRIVILEGES` key word in `ALL PRIVILEGES` is required. SQL does not support setting the privileges on more than one table per command.

The SQL92 syntax for GRANT allows setting privileges for individual columns within a table, and allows setting a privilege to grant the same privileges to others:

```
GRANT privilege [, ...]
    ON object [ ( column [, ...] ) ] [, ...]
    TO { PUBLIC | username [, ...] } [ WITH GRANT OPTION ]
```

SQL allows to grant the USAGE privilege on other kinds of objects: CHARACTER SET, COLLATION, TRANSLATION, DOMAIN.

The TRIGGER privilege was introduced in SQL99. The RULE privilege is a PostgreSQL extension.

See Also

REVOKE

Name

INSERT — create new rows in a table

Synopsis

<refsynopsisdivinfo>2000-08-08</refsynopsisdivinfo>

```
INSERT INTO table [ ( column [, ...] ) ]
    { DEFAULT VALUES | VALUES ( expression [, ...] ) | SELECT query }
```

Inputs

table

The name of an existing table.

column

The name of a column in *table*.

DEFAULT VALUES

All columns will be filled by NULLs or by values specified when the table was created using DEFAULT clauses.

expression

A valid expression or value to assign to *column*.

query

A valid query. Refer to the SELECT statement for a further description of valid arguments.

Outputs

INSERT *oid* 1

Message returned if only one row was inserted. *oid* is the numeric OID of the inserted row.

INSERT 0 #

Message returned if more than one rows were inserted. # is the number of rows inserted.

Description

INSERT allows one to insert new rows into a table. One can insert a single row at a time or several rows as a result of a query. The columns in the target list may be listed in any order.

Each column not present in the target list will be inserted using a default value, either a declared DEFAULT value or NULL. PostgreSQL will reject the new column if a NULL is inserted into a column declared NOT NULL.

If the expression for each column is not of the correct data type, automatic type coercion will be attempted.

You must have insert privilege to a table in order to append to it, as well as select privilege on any table specified in a WHERE clause.

Usage

Insert a single row into table `films`:

```
INSERT INTO films VALUES
    ('UA502','Bananas',105,'1971-07-13','Comedy',INTERVAL '82
 minute');
```

In this second example the last column `len` is omitted and therefore it will have the default value of NULL:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DATE '1961-06-16', 'Drama');
```

Insert a single row into table distributors; note that only column `name` is specified, so the omitted column `did` will be assigned its default value:

```
INSERT INTO distributors (name) VALUES ('British Lion');
```

Insert several rows into table films from table `tmp`:

```
INSERT INTO films SELECT * FROM tmp;
```

Insert into arrays (refer to the *PostgreSQL User's Guide* for further information about arrays):

```
-- Create an empty 3x3 gameboard for noughts-and-crosses
-- (all of these queries create the same board attribute)
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1,'{{"","",""},{},{"",""}}');
INSERT INTO tictactoe (game, board[3][3])
    VALUES (2,'{}');
INSERT INTO tictactoe (game, board)
    VALUES (3,'{{,,},{,,},{,,}}');
```

Compatibility

SQL92

`INSERT` is fully compatible with SQL92. Possible limitations in features of the *`query`* clause are documented for SELECT.

Name

LISTEN — listen for a notification

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
LISTEN name
```

Inputs

name

Name of notify condition.

Outputs

`LISTEN`

Message returned upon successful completion of registration.

`NOTICE Async_Listen: We are already listening on name`

If this backend is already registered for that notify condition.

Description

`LISTEN` registers the current PostgreSQL backend as a listener on the notify condition *name*.

Whenever the command `NOTIFY` *name* is invoked, either by this backend or another one connected to the same database, all the backends currently listening on that notify condition are notified, and each will in turn notify its connected frontend application. See the discussion of `NOTIFY` for more information.

A backend can be unregistered for a given notify condition with the `UNLISTEN` command. Also, a backend's listen registrations are automatically cleared when the backend process exits.

The method a frontend application must use to detect notify events depends on which PostgreSQL application programming interface it uses. With the basic libpq library, the application issues `LISTEN` as an ordinary SQL command, and then must periodically call the routine `PQnotifies` to find out whether any notify events have been received. Other interfaces such as libpgtcl provide higher-level methods for handling notify events; indeed, with libpgtcl the application programmer should not even issue `LISTEN` or `UNLISTEN` directly. See the documentation for the library you are using for more details.

NOTIFY contains a more extensive discussion of the use of `LISTEN` and `NOTIFY`.

Notes

name can be any string valid as a name; it need not correspond to the name of any actual table. If *notifyname* is enclosed in double-quotes, it need not even be a syntactically valid name, but can be any string up to 31 characters long.

In some previous releases of PostgreSQL, *name* had to be enclosed in double-quotes when it did not correspond to any existing table name, even if syntactically valid as a name. That is no longer required.

Usage

Configure and execute a listen/notify sequence from psql:

```
LISTEN virtual;
NOTIFY virtual;

Asynchronous NOTIFY 'virtual' from backend with pid '8448' received.
```

Compatibility

SQL92

There is no `LISTEN` in SQL92.

Name

LOAD — load or reload a shared library file

Synopsis

```
LOAD 'filename'
```

Description

Loads a shared library file into the PostgreSQL backend's address space. If the file had been loaded previously, it is first unloaded. This command is primarily useful to unload and reload a shared library file that has been changed since the backend first loaded it. To make use of the shared library, function(s) in it need to be declared using the CREATE FUNCTION command.

The file name is specified in the same way as for shared library names in CREATE FUNCTION; in particular, one may rely on a search path and automatic addition of the system's standard shared library file name extension. See the *Programmer's Guide* for more detail.

Compatibility

`LOAD` is a PostgreSQL extension.

See Also

CREATE FUNCTION, *PostgreSQL Programmer's Guide*

Name

LOCK — explicitly lock a table

Synopsis

<refsynopsisdivinfo>2001-07-09</refsynopsisdivinfo>

```
LOCK [ TABLE ] name [, ...]
LOCK [ TABLE ] name [, ...] IN lockmode MODE

where lockmode is one of:

 ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE |
 SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

Inputs

name

The name of an existing table to lock.

ACCESS SHARE MODE

Note

This lock mode is acquired automatically over tables being queried.

This is the least restrictive lock mode. It conflicts only with ACCESS EXCLUSIVE mode. It is used to protect a table from being modified by concurrent `ALTER TABLE`, `DROP TABLE` and `VACUUM FULL` commands.

ROW SHARE MODE

Note

Automatically acquired by `SELECT ... FOR UPDATE`.

Conflicts with EXCLUSIVE and ACCESS EXCLUSIVE lock modes.

ROW EXCLUSIVE MODE

Note

Automatically acquired by `UPDATE`, `DELETE`, and `INSERT` statements.

Conflicts with SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes.

SHARE UPDATE EXCLUSIVE MODE

Note

Automatically acquired by `VACUUM` (without `FULL`).

Conflicts with SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes. This mode protects a table against concurrent schema changes and VACUUMs.

SHARE MODE

Note

Automatically acquired by `CREATE INDEX`. Share-locks the entire table.

Conflicts with ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes. This mode protects a table against concurrent data updates.

SHARE ROW EXCLUSIVE MODE

Note

This is like EXCLUSIVE MODE, but allows ROW SHARE locks by others.

Conflicts with ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes.

EXCLUSIVE MODE

Note

This mode is yet more restrictive than SHARE ROW EXCLUSIVE. It blocks all concurrent ROW SHARE/SELECT...FOR UPDATE queries.

Conflicts with ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes. This mode allows only concurrent ACCESS SHARE, i.e., only reads from the table can proceed in parallel with a transaction holding this lock mode.

ACCESS EXCLUSIVE MODE

Note

Automatically acquired by `ALTER TABLE`, `DROP TABLE`, `VACUUM FULL` statements. This is the most restrictive lock mode which protects a locked table from any concurrent operations.

Note

This lock mode is also acquired by an unqualified `LOCK TABLE` (i.e., the command without an explicit lock mode option).

Conflicts with all lock modes.

Outputs

`LOCK TABLE`

The lock was successfully acquired.

`ERROR` *`name`*`: Table does not exist.`

Message returned if *`name`* does not exist.

Description

`LOCK TABLE` controls concurrent access to a table for the duration of a transaction. PostgreSQL always uses the least restrictive lock mode whenever possible. `LOCK TABLE` provides for cases when you might need more restrictive locking.

RDBMS locking uses the following terminology:

EXCLUSIVE

An exclusive lock prevents other locks of the same type from being granted. (Note: ROW EXCLUSIVE mode does not follow this naming convention perfectly, since it is shared at the level of the table; it is exclusive only with respect to specific rows that are being updated.)

SHARE

A shared lock allows others to also hold the same type of lock, but prevents the corresponding EXCLUSIVE lock from being granted.

ACCESS

Locks table schema.

ROW

Locks individual rows.

For example, suppose an application runs a transaction at READ COMMITTED isolation level and needs to ensure the existence of data in a table for the duration of the transaction. To achieve this you could obtain SHARE lock mode over the table before querying. This will prevent concurrent data changes and ensure further read operations over the table see data in their actual current state, because SHARE lock mode conflicts with any ROW EXCLUSIVE lock acquired by writers, and your `LOCK TABLE` *`name`* `IN SHARE MODE` statement will wait until any concurrent write operations commit or rollback. Thus, once you obtain the lock, there are no uncommitted writes outstanding.

Note

To read data in their actual current state when running a transaction at the SERIALIZABLE isolation level, you have to execute the LOCK TABLE statement before executing any DML statement. A serializable transaction's view of data will be frozen when its first DML statement begins.

In addition to the requirements above, if a transaction is going to change data in a table, then SHARE ROW EXCLUSIVE lock mode should be acquired to prevent deadlock conditions when two concurrent transactions attempt to lock the table in SHARE mode and then try to change data in this table, both (implicitly) acquiring ROW EXCLUSIVE lock mode that conflicts with a concurrent SHARE lock.

To continue with the deadlock (when two transactions wait for one another) issue raised above, you should follow two general rules to prevent deadlock conditions:

- Transactions have to acquire locks on the same objects in the same order.

  For example, if one application updates row R1 and than updates row R2 (in the same transaction) then the second application shouldn't update row R2 if it's going to update row R1 later (in a single transaction). Instead, it should update rows R1 and R2 in the same order as the first application.

- Transactions should acquire two conflicting lock modes only if one of them is self-conflicting (i.e., may be held by only one transaction at a time). If multiple lock modes are involved, then transactions should always acquire the most restrictive mode first.

  An example for this rule was given previously when discussing the use of SHARE ROW EXCLUSIVE mode rather than SHARE mode.

  ### Note

  PostgreSQL does detect deadlocks and will rollback at least one waiting transaction to resolve the deadlock.

When locking multiple tables, the command LOCK a, b; is equivalent to LOCK a; LOCK b;. The tables are locked one-by-one in the order specified in the `LOCK` command.

Notes

`LOCK ... IN ACCESS SHARE MODE` requires `SELECT` privileges on the target table. All other forms of `LOCK` require `UPDATE` and/or `DELETE` privileges.

`LOCK` is useful only inside a transaction block (`BEGIN...COMMIT`), since the lock is dropped as soon as the transaction ends. A `LOCK` command appearing outside any transaction block forms a self-contained transaction, so the lock will be dropped as soon as it is obtained.

Usage

Illustrate a SHARE lock on a primary key table when going to perform inserts into a foreign key table:

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- Do ROLLBACK if record was not returned
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

Take a SHARE ROW EXCLUSIVE lock on a primary key table when going to perform a delete operation:

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

Compatibility

SQL92

There is no `LOCK TABLE` in SQL92, which instead uses `SET TRANSACTION` to specify concurrency levels on transactions. We support that too; see SET TRANSACTION for details.

Except for ACCESS SHARE, ACCESS EXCLUSIVE, and SHARE UPDATE EXCLUSIVE lock modes, the PostgreSQL lock modes and the `LOCK TABLE` syntax are compatible with those present in Oracle(TM).

Name

MOVE — position a cursor on a specified row of a table

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
MOVE [ direction ] [ count ]
    { IN | FROM } cursor
```

Description

`MOVE` allows a user to move cursor position a specified number of rows. `MOVE` works like the `FETCH` command, but only positions the cursor and does not return rows.

Refer to FETCH for details on syntax and usage.

Notes

`MOVE` is a PostgreSQL language extension.

Refer to FETCH for a description of valid arguments. Refer to DECLARE to define a cursor. Refer to BEGIN , COMMIT , and ROLLBACK for further information about transactions.

Usage

Set up and use a cursor:

```
BEGIN WORK;
DECLARE liahona CURSOR  FOR SELECT * FROM films;
-- Skip first 5 rows:
MOVE FORWARD 5 IN liahona;
MOVE
-- Fetch 6th row in the cursor liahona:
FETCH 1 IN liahona;
FETCH

 code  | title  | did | date_prod | kind   | len
-------+--------+-----+-----------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22| Action | 01:37
(1 row)
-- close the cursor liahona and commit work:
CLOSE liahona;
COMMIT WORK;
```

Compatibility

SQL92

There is no SQL92 `MOVE` statement. Instead, SQL92 allows one to `FETCH` rows from an absolute cursor position, implicitly moving the cursor to the correct position.

Name

NOTIFY — generate a notification

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
NOTIFY name
```

Inputs

notifyname

Notify condition to be signaled.

Outputs

NOTIFY

Acknowledgement that notify command has executed.

Notify events

Events are delivered to listening frontends; whether and how each frontend application reacts depends on its programming.

Description

The NOTIFY command sends a notify event to each frontend application that has previously executed LISTEN *notifyname* for the specified notify condition in the current database.

The information passed to the frontend for a notify event includes the notify condition name and the notifying backend process's PID. It is up to the database designer to define the condition names that will be used in a given database and what each one means.

Commonly, the notify condition name is the same as the name of some table in the database, and the notify event essentially means “I changed this table, take a look at it to see what's new”. But no such association is enforced by the NOTIFY and LISTEN commands. For example, a database designer could use several different condition names to signal different sorts of changes to a single table.

NOTIFY provides a simple form of signal or IPC (interprocess communication) mechanism for a collection of processes accessing the same PostgreSQL database. Higher-level mechanisms can be built by using tables in the database to pass additional data (beyond a mere condition name) from notifier to listener(s).

When NOTIFY is used to signal the occurrence of changes to a particular table, a useful programming technique is to put the NOTIFY in a rule that is triggered by table updates. In this way, notification happens automatically when the table is changed, and the application programmer can't accidentally forget to do it.

NOTIFY interacts with SQL transactions in some important ways. Firstly, if a NOTIFY is executed inside a transaction, the notify events are not delivered until and unless the transaction is committed. This is appropriate, since if the transaction is aborted we would like all the commands within it to have had no effect, including NOTIFY. But it can be disconcerting if one is expecting the notify events to be delivered immediately. Secondly, if a listening backend receives a notify signal while it is within a transaction, the notify event will not be delivered to its connected frontend until just after the transaction is completed

(either committed or aborted). Again, the reasoning is that if a notify were delivered within a transaction that was later aborted, one would want the notification to be undone somehow---but the backend cannot "take back" a notify once it has sent it to the frontend. So notify events are only delivered between transactions. The upshot of this is that applications using NOTIFY for real-time signaling should try to keep their transactions short.

NOTIFY behaves like Unix signals in one important respect: if the same condition name is signaled multiple times in quick succession, recipients may get only one notify event for several executions of NOTIFY. So it is a bad idea to depend on the number of notifies received. Instead, use NOTIFY to wake up applications that need to pay attention to something, and use a database object (such as a sequence) to keep track of what happened or how many times it happened.

It is common for a frontend that sends NOTIFY to be listening on the same notify name itself. In that case it will get back a notify event, just like all the other listening frontends. Depending on the application logic, this could result in useless work---for example, re-reading a database table to find the same updates that that frontend just wrote out. In PostgreSQL 6.4 and later, it is possible to avoid such extra work by noticing whether the notifying backend process's PID (supplied in the notify event message) is the same as one's own backend's PID (available from libpq). When they are the same, the notify event is one's own work bouncing back, and can be ignored. (Despite what was said in the preceding paragraph, this is a safe technique. PostgreSQL keeps self-notifies separate from notifies arriving from other backends, so you cannot miss an outside notify by ignoring your own notifies.)

Notes

name can be any string valid as a name; it need not correspond to the name of any actual table. If *name* is enclosed in double-quotes, it need not even be a syntactically valid name, but can be any string up to 31 characters long.

In some previous releases of PostgreSQL, *name* had to be enclosed in double-quotes when it did not correspond to any existing table name, even if syntactically valid as a name. That is no longer required.

In PostgreSQL releases prior to 6.4, the backend PID delivered in a notify message was always the PID of the frontend's own backend. So it was not possible to distinguish one's own notifies from other clients' notifies in those earlier releases.

Usage

Configure and execute a listen/notify sequence from psql:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous NOTIFY 'virtual' from backend with pid '8448' received.
```

Compatibility

SQL92

There is no NOTIFY statement in SQL92.

Name

REINDEX — rebuild corrupted indexes

Synopsis

<refsynopsisdivinfo>2000-03-30</refsynopsisdivinfo>

```
REINDEX { TABLE | DATABASE | INDEX } name [ FORCE ]
```

Inputs

TABLE

Recreate all indexes of a specified table.

DATABASE

Recreate all system indexes of a specified database. (User-table indexes are not included.)

INDEX

Recreate a specified index.

name

The name of the specific table/database/index to be be reindexed.

FORCE

Force rebuild of system indexes. Without this keyword `REINDEX` skips system indexes that are not marked invalid. FORCE is irrelevant for `REINDEX INDEX`, or when reindexing user indexes.

Outputs

`REINDEX`

Message returned if the table is successfully reindexed.

Description

`REINDEX` is used to rebuild corrupted indexes. Although in theory this should never be necessary, in practice indexes may become corrupted due to software bugs or hardware failures. `REINDEX` provides a recovery method.

If you suspect corruption of an index on a user table, you can simply rebuild that index, or all indexes on the table, using `REINDEX INDEX` or `REINDEX TABLE`.

Note

Another approach to dealing with a corrupted user-table index is just to drop and recreate it. This may in fact be preferable if you would like to maintain some semblance of normal operation on the table meanwhile. `REINDEX` acquires exclusive lock on the table, while `CREATE INDEX` only locks out writes not reads of the table.

Things are more difficult if you need to recover from corruption of an index on a system table. In this case it's important for the backend doing the recovery to not have used any of the suspect indexes itself. (Indeed, in this sort of scenario you may find that backends are crashing immediately at startup, due to reliance on the corrupted indexes.) To recover safely, the postmaster must be shut down and a stand-alone PostgreSQL backend must be started instead, giving it the command-line options -O and -P (these options allow system table modifications and prevent use of system indexes, respectively). Then issue `REINDEX INDEX`, `REINDEX TABLE`, or `REINDEX DATABASE` depending on how much you want to reconstruct. If in doubt, use `REINDEX DATABASE FORCE` to force reconstruction of all system indexes in the database. Then quit the standalone backend and restart the postmaster.

Since this is likely the only situation when most people will ever use a standalone backend, some usage notes might be in order:

- Start the backend with a command like

  postgres -D $PGDATA -O -P my_database

  Provide the correct path to the database area with `-D`, or make sure that the environment variable `PGDATA` is set. Also specify the name of the particular database you want to work in.

- You can issue any SQL command, not only `REINDEX`.

- Be aware that the standalone backend treats newline as the command entry terminator; there is no intelligence about semicolons, as there is in psql. To continue a command across multiple lines, you must type backslash just before each newline except the last one. Also, you won't have any of the conveniences of readline processing (no command history, for example).

- To quit the backend, type EOF (control-D, usually).

See the postgres reference page for more information.

Usage

Recreate the indexes on the table `mytable`:

```
REINDEX TABLE mytable;
```

Rebuild a single index:

```
REINDEX INDEX my_index;
```

Rebuild all system indexes (this will only work in a standalone backend):

```
REINDEX DATABASE my_database FORCE;
```

Compatibility

SQL92

There is no `REINDEX` in SQL92.

Name

RESET — restore the value of a run-time parameter to a default value

Synopsis

```
RESET variable

RESET ALL
```

Inputs

variable

The name of a run-time parameter. See SET for a list.

ALL

Resets all run-time parameters to default values.

Description

`RESET` restores run-time parameters to their default values. Refer to SET for details. `RESET` is an alternate form for

```
SET variable TO DEFAULT
```

Diagnostics

See under the SET command.

Examples

Set DateStyle to its default value:

```
RESET DateStyle;
```

Set Geqo to its default value:

```
RESET GEQO;
```

Compatibility

`RESET` is a PostgreSQL extension.

Name

REVOKE — remove access privileges

Synopsis

```
REVOKE { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES |
 TRIGGER } [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] object [, ...]
    FROM { username | GROUP groupname | PUBLIC } [, ...]
```

Description

`REVOKE` allows the creator of an object to revoke previously granted permissions from one or more users or groups of users. The key word `PUBLIC` refers to the implicitly defined group of all users.

Note that any particular user will have the sum of privileges granted directly to him, privileges granted to any group he is presently a member of, and privileges granted to `PUBLIC`. Thus, for example, revoking SELECT privilege from `PUBLIC` does not necessarily mean that all users have lost SELECT privilege on the object: those who have it granted directly or via a group will still have it.

See the description of the GRANT command for the meaning of the privilege types.

Notes

Use psql's `\z` command to display the privileges granted on existing objects. See also GRANT for information about the format.

Examples

Revoke insert privilege for the public on table `films`:

```
REVOKE INSERT ON films FROM PUBLIC;
```

Revoke all privileges from user `manuel` on view `kinds`:

```
REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

Compatibility

SQL92

The compatibility notes of the GRANT command apply analogously to `REVOKE`. The syntax summary is:

```
REVOKE [ GRANT OPTION FOR ] { SELECT | INSERT | UPDATE | DELETE |
 REFERENCES }
    ON object [ ( column [, ...] ) ]
    FROM { PUBLIC | username [, ...] }
    { RESTRICT | CASCADE }
```

If user1 gives a privilege WITH GRANT OPTION to user2, and user2 gives it to user3 then user1 can revoke this privilege in cascade using the CASCADE keyword. If user1 gives a privilege WITH GRANT

OPTION to user2, and user2 gives it to user3, then if user1 tries to revoke this privilege it fails if he specifies the RESTRICT keyword.

See Also

GRANT

Name

ROLLBACK — abort the current transaction

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
ROLLBACK [ WORK | TRANSACTION ]
```

Inputs

None.

Outputs

`ABORT`

Message returned if successful.

```
NOTICE: ROLLBACK: no transaction in progress
```

If there is not any transaction currently in progress.

Description

`ROLLBACK` rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Notes

Use COMMIT to successfully terminate a transaction. ABORT is a synonym for `ROLLBACK`.

Usage

To abort all changes:

```
ROLLBACK WORK;
```

Compatibility

SQL92

SQL92 only specifies the two forms `ROLLBACK` and `ROLLBACK WORK`. Otherwise full compatibility.

Name

SELECT — retrieve rows from a table or view

Synopsis

<refsynopsisdivinfo>2000-12-11</refsynopsisdivinfo>

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ FOR UPDATE [ OF tablename [, ...] ] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]

where from_item can be:

[ ONLY ] table_name [ * ]
    [ [ AS ] alias [ ( column_alias_list ) ] ]
|
( select )
    [ AS ] alias [ ( column_alias_list ) ]
|
from_item [ NATURAL ] join_type from_item
    [ ON join_condition | USING ( join_column_list ) ]
```

Inputs

expression

The name of a table's column or an expression.

output_name

Specifies another name for an output column using the AS clause. This name is primarily used to label the column for display. It can also be used to refer to the column's value in ORDER BY and GROUP BY clauses. But the *output_name* cannot be used in the WHERE or HAVING clauses; write out the expression instead.

from_item

A table reference, sub-SELECT, or JOIN clause. See below for details.

condition

A boolean expression giving a result of true or false. See the WHERE and HAVING clause descriptions below.

select

A select statement with all features except the ORDER BY, FOR UPDATE, and LIMIT clauses (even those can be used when the select is parenthesized).

FROM items can contain:

table_name

The name of an existing table or view. If ONLY is specified, only that table is scanned. If ONLY is not specified, the table and all its descendant tables (if any) are scanned. * can be appended to the table name to indicate that descendant tables are to be scanned, but in the current version, this is the default behavior. (In releases before 7.1, ONLY was the default behavior.)

alias

A substitute name for the preceding *table_name*. An alias is used for brevity or to eliminate ambiguity for self-joins (where the same table is scanned multiple times). If an alias is written, a column alias list can also be written to provide substitute names for one or more columns of the table.

select

A sub-SELECT can appear in the FROM clause. This acts as though its output were created as a temporary table for the duration of this single SELECT command. Note that the sub-SELECT must be surrounded by parentheses, and an alias *must* be provided for it.

join_type

One of `[ INNER ] JOIN`, `LEFT [ OUTER ] JOIN`, `RIGHT [ OUTER ] JOIN`, `FULL [ OUTER ] JOIN`, or `CROSS JOIN`. For INNER and OUTER join types, exactly one of NATURAL, ON *join_condition*, or USING (*join_column_list*) must appear. For CROSS JOIN, none of these items may appear.

join_condition

A qualification condition. This is similar to the WHERE condition except that it only applies to the two from_items being joined in this JOIN clause.

join_column_list

A USING column list (a, b, ...) is shorthand for the ON condition left_table.a = right_table.a AND left_table.b = right_table.b ...

Outputs

Rows

The complete set of rows resulting from the query specification.

count

The count of rows returned by the query.

Description

`SELECT` will return rows from one or more tables. Candidates for selection are rows which satisfy the WHERE condition; if WHERE is omitted, all rows are candidates. (See WHERE Clause .)

Actually, the returned rows are not directly the rows produced by the FROM/WHERE/GROUP BY/HAVING clauses; rather, the output rows are formed by computing the SELECT output expressions for

each selected row. * can be written in the output list as a shorthand for all the columns of the selected rows. Also, one can write *table_name*.* as a shorthand for the columns coming from just that table.

DISTINCT will eliminate duplicate rows from the result. ALL (the default) will return all candidate rows, including duplicates.

DISTINCT ON eliminates rows that match on all the specified expressions, keeping only the first row of each set of duplicates. The DISTINCT ON expressions are interpreted using the same rules as for ORDER BY items; see below. Note that the "first row" of each set is unpredictable unless ORDER BY is used to ensure that the desired row appears first. For example,

```
SELECT DISTINCT ON (location) location, time, report
FROM weatherReports
ORDER BY location, time DESC;
```

retrieves the most recent weather report for each location. But if we had not used ORDER BY to force descending order of time values for each location, we'd have gotten a report of unpredictable age for each location.

The GROUP BY clause allows a user to divide a table into groups of rows that match on one or more values. (See GROUP BY Clause .)

The HAVING clause allows selection of only those groups of rows meeting the specified condition. (See HAVING Clause .)

The ORDER BY clause causes the returned rows to be sorted in a specified order. If ORDER BY is not given, the rows are returned in whatever order the system finds cheapest to produce. (See ORDER BY Clause .)

SELECT queries can be combined using UNION, INTERSECT, and EXCEPT operators. Use parentheses if necessary to determine the ordering of these operators.

The UNION operator computes the collection of rows returned by the queries involved. Duplicate rows are eliminated unless ALL is specified. (See UNION Clause .)

The INTERSECT operator computes the rows that are common to both queries. Duplicate rows are eliminated unless ALL is specified. (See INTERSECT Clause .)

The EXCEPT operator computes the rows returned by the first query but not the second query. Duplicate rows are eliminated unless ALL is specified. (See EXCEPT Clause .)

The FOR UPDATE clause allows the SELECT statement to perform exclusive locking of selected rows.

The LIMIT clause allows a subset of the rows produced by the query to be returned to the user. (See LIMIT Clause .)

You must have SELECT privilege to a table to read its values (See the GRANT/REVOKE statements).

FROM Clause

The FROM clause specifies one or more source tables for the SELECT. If multiple sources are specified, the result is conceptually the Cartesian product of all the rows in all the sources --- but usually qualification conditions are added to restrict the returned rows to a small subset of the Cartesian product.

When a FROM item is a simple table name, it implicitly includes rows from sub-tables (inheritance children) of the table. `ONLY` will suppress rows from sub-tables of the table. Before PostgreSQL 7.1, this was the default result, and adding sub-tables was done by appending `*` to the table name. This old behaviour is available via the command `SET SQL_Inheritance TO OFF;`

A FROM item can also be a parenthesized sub-SELECT (note that an alias clause is required for a sub-SELECT!). This is an extremely handy feature since it's the only way to get multiple levels of grouping, aggregation, or sorting in a single query.

Finally, a FROM item can be a JOIN clause, which combines two simpler FROM items. (Use parentheses if necessary to determine the order of nesting.)

A CROSS JOIN or INNER JOIN is a simple Cartesian product, the same as you get from listing the two items at the top level of FROM. CROSS JOIN is equivalent to INNER JOIN ON (TRUE), that is, no rows are removed by qualification. These join types are just a notational convenience, since they do nothing you couldn't do with plain FROM and WHERE.

LEFT OUTER JOIN returns all rows in the qualified Cartesian product (i.e., all combined rows that pass its ON condition), plus one copy of each row in the left-hand table for which there was no right-hand row that passed the ON condition. This left-hand row is extended to the full width of the joined table by inserting NULLs for the right-hand columns. Note that only the JOIN's own ON or USING condition is considered while deciding which rows have matches. Outer ON or WHERE conditions are applied afterwards.

Conversely, RIGHT OUTER JOIN returns all the joined rows, plus one row for each unmatched right-hand row (extended with nulls on the left). This is just a notational convenience, since you could convert it to a LEFT OUTER JOIN by switching the left and right inputs.

FULL OUTER JOIN returns all the joined rows, plus one row for each unmatched left-hand row (extended with nulls on the right), plus one row for each unmatched right-hand row (extended with nulls on the left).

For all the JOIN types except CROSS JOIN, you must write exactly one of ON *join_condition*, USING (*join_column_list*), or NATURAL. ON is the most general case: you can write any qualification expression involving the two tables to be joined. A USING column list (a, b, ...) is shorthand for the ON condition left_table.a = right_table.a AND left_table.b = right_table.b ... Also, USING implies that only one of each pair of equivalent columns will be included in the JOIN output, not both. NATURAL is shorthand for a USING list that mentions all similarly-named columns in the tables.

WHERE Clause

The optional WHERE condition has the general form:

`WHERE` *boolean_expr*

boolean_expr can consist of any expression which evaluates to a boolean value. In many cases, this expression will be:

expr cond_op expr

or

log_op expr

where *cond_op* can be one of: =, <, <=, >, >= or <>, a conditional operator like ALL, ANY, IN, LIKE, or a locally defined operator, and *log_op* can be one of: AND, OR, NOT. SELECT will ignore all rows for which the WHERE condition does not return TRUE.

GROUP BY Clause

GROUP BY specifies a grouped table derived by the application of this clause:

```
GROUP BY expression [, ...]
```

GROUP BY will condense into a single row all selected rows that share the same values for the grouped columns. Aggregate functions, if any, are computed across all rows making up each group, producing a separate value for each group (whereas without GROUP BY, an aggregate produces a single value computed across all the selected rows). When GROUP BY is present, it is not valid for the SELECT output expression(s) to refer to ungrouped columns except within aggregate functions, since there would be more than one possible value to return for an ungrouped column.

A GROUP BY item can be an input column name, or the name or ordinal number of an output column (SELECT expression), or it can be an arbitrary expression formed from input-column values. In case of ambiguity, a GROUP BY name will be interpreted as an input-column name rather than an output column name.

HAVING Clause

The optional HAVING condition has the general form:

```
HAVING boolean_expr
```

where *boolean_expr* is the same as specified for the WHERE clause.

HAVING specifies a grouped table derived by the elimination of group rows that do not satisfy the *boolean_expr*. HAVING is different from WHERE: WHERE filters individual rows before application of GROUP BY, while HAVING filters group rows created by GROUP BY.

Each column referenced in *boolean_expr* shall unambiguously reference a grouping column, unless the reference appears within an aggregate function.

ORDER BY Clause

```
ORDER BY expression [ ASC | DESC | USING operator ] [, ...]
```

An ORDER BY item can be the name or ordinal number of an output column (SELECT expression), or it can be an arbitrary expression formed from input-column values. In case of ambiguity, an ORDER BY name will be interpreted as an output-column name.

The ordinal number refers to the ordinal (left-to-right) position of the result column. This feature makes it possible to define an ordering on the basis of a column that does not have a proper name. This is never absolutely necessary because it is always possible to assign a name to a result column using the AS clause, e.g.:

```
SELECT title, date_prod + 1 AS newlen FROM films ORDER BY newlen;
```

It is also possible to ORDER BY arbitrary expressions (an extension to SQL92), including fields that do not appear in the SELECT result list. Thus the following statement is legal:

```
SELECT name FROM distributors ORDER BY code;
```

A limitation of this feature is that an ORDER BY clause applying to the result of a UNION, INTERSECT, or EXCEPT query may only specify an output column name or number, not an expression.

Note that if an ORDER BY item is a simple name that matches both a result column name and an input column name, ORDER BY will interpret it as the result column name. This is the opposite of the choice that GROUP BY will make in the same situation. This inconsistency is mandated by the SQL92 standard.

Optionally one may add the keyword DESC (descending) or ASC (ascending) after each column name in the ORDER BY clause. If not specified, ASC is assumed by default. Alternatively, a specific ordering operator name may be specified. ASC is equivalent to USING < and DESC is equivalent to USING >.

The null value sorts higher than any other value in a domain. In other words, with ascending sort order nulls sort at the end and with descending sort order nulls sort at the beginning.

UNION Clause

```
table_query UNION [ ALL ] table_query
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
```

where *table_query* specifies any select expression without an ORDER BY, FOR UPDATE, or LIMIT clause. (ORDER BY and LIMIT can be attached to a sub-expression if it is enclosed in parentheses. Without parentheses, these clauses will be taken to apply to the result of the UNION, not to its right-hand input expression.)

The UNION operator computes the collection (set union) of the rows returned by the queries involved. The two SELECTs that represent the direct operands of the UNION must produce the same number of columns, and corresponding columns must be of compatible data types.

The result of UNION does not contain any duplicate rows unless the ALL option is specified. ALL prevents elimination of duplicates.

Multiple UNION operators in the same SELECT statement are evaluated left to right, unless otherwise indicated by parentheses.

Currently, FOR UPDATE may not be specified either for a UNION result or for the inputs of a UNION.

INTERSECT Clause

```
table_query INTERSECT [ ALL ] table_query
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
```

where *table_query* specifies any select expression without an ORDER BY, FOR UPDATE, or LIMIT clause.

INTERSECT is similar to UNION, except that it produces only rows that appear in both query outputs, rather than rows that appear in either.

The result of INTERSECT does not contain any duplicate rows unless the ALL option is specified. With ALL, a row that has m duplicates in L and n duplicates in R will appear min(m,n) times.

Multiple INTERSECT operators in the same SELECT statement are evaluated left to right, unless parentheses dictate otherwise. INTERSECT binds more tightly than UNION --- that is, A UNION B INTERSECT C will be read as A UNION (B INTERSECT C) unless otherwise specified by parentheses.

EXCEPT Clause

```
table_query EXCEPT [ ALL ] table_query
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
```

where *table_query* specifies any select expression without an ORDER BY, FOR UPDATE, or LIMIT clause.

EXCEPT is similar to UNION, except that it produces only rows that appear in the left query's output but not in the right query's output.

The result of EXCEPT does not contain any duplicate rows unless the ALL option is specified. With ALL, a row that has m duplicates in L and n duplicates in R will appear max(m-n,0) times.

Multiple EXCEPT operators in the same SELECT statement are evaluated left to right, unless parentheses dictate otherwise. EXCEPT binds at the same level as UNION.

LIMIT Clause

```
LIMIT { count | ALL }
OFFSET start
```

where *count* specifies the maximum number of rows to return, and *start* specifies the number of rows to skip before starting to return rows.

LIMIT allows you to retrieve just a portion of the rows that are generated by the rest of the query. If a limit count is given, no more than that many rows will be returned. If an offset is given, that many rows will be skipped before starting to return rows.

When using LIMIT, it is a good idea to use an ORDER BY clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows---you may be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? You don't know what ordering unless you specify ORDER BY.

As of PostgreSQL 7.0, the query optimizer takes LIMIT into account when generating a query plan, so you are very likely to get different plans (yielding different row orders) depending on what you use for LIMIT and OFFSET. Thus, using different LIMIT/OFFSET values to select different subsets of a query result *will give inconsistent results* unless you enforce a predictable result ordering with ORDER BY. This is not a bug; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless ORDER BY is used to constrain the order.

Usage

To join the table `films` with the table `distributors`:

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d, films f
    WHERE f.did = d.did

           title           | did |   name           | date_prod  |
 kind
---------------------------+-----+------------------+------------
+----------
 The Third Man             | 101 | British Lion     | 1949-12-23 |
 Drama
 The African Queen         | 101 | British Lion     | 1951-08-11 |
 Romantic
 Une Femme est une Femme   | 102 | Jean Luc Godard  | 1961-03-12 |
 Romantic
 Vertigo                   | 103 | Paramount        | 1958-11-14 |
 Action
 Becket                    | 103 | Paramount        | 1964-02-03 |
 Drama
 48 Hrs                    | 103 | Paramount        | 1982-10-22 |
 Action
 War and Peace             | 104 | Mosfilm          | 1967-02-12 |
 Drama
 West Side Story           | 105 | United Artists   | 1961-01-03 |
 Musical
 Bananas                   | 105 | United Artists   | 1971-07-13 |
 Comedy
 Yojimbo                   | 106 | Toho             | 1961-06-16 |
 Drama
 There's a Girl in my Soup | 107 | Columbia         | 1970-06-11 |
 Comedy
 Taxi Driver               | 107 | Columbia         | 1975-05-15 |
 Action
 Absence of Malice         | 107 | Columbia         | 1981-11-15 |
 Action
 Storia di una donna       | 108 | Westward         | 1970-08-15 |
 Romantic
 The King and I            | 109 | 20th Century Fox | 1956-08-11 |
 Musical
 Das Boot                  | 110 | Bavaria Atelier  | 1981-11-11 |
 Drama
 Bed Knobs and Broomsticks | 111 | Walt Disney      |            |
 Musical
(17 rows)
```

To sum the column `len` of all films and group the results by `kind`:

```
SELECT kind, SUM(len) AS total FROM films GROUP BY kind;

   kind   | total
----------+-------
```

```
 Action   | 07:34
 Comedy   | 02:58
 Drama    | 14:28
 Musical  | 06:42
 Romantic | 04:38
(5 rows)
```

To sum the column `len` of all films, group the results by `kind` and show those group totals that are less than 5 hours:

```
SELECT kind, SUM(len) AS total
    FROM films
    GROUP BY kind
    HAVING SUM(len) < INTERVAL '5 hour';

 kind     | total
----------+-------
 Comedy   | 02:58
 Romantic | 04:38
(2 rows)
```

The following two examples are identical ways of sorting the individual results according to the contents of the second column (`name`):

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;

 did |       name
-----+------------------
 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
(13 rows)
```

This example shows how to obtain the union of the tables `distributors` and `actors`, restricting the results to those that begin with letter W in each table. Only distinct rows are wanted, so the ALL keyword is omitted:

```
distributors:               actors:
 did |     name             id |     name
-----+--------------       ----+----------------
 108 | Westward               1 | Woody Allen
 111 | Walt Disney            2 | Warren Beatty
 112 | Warner Bros.           3 | Walter Matthau
 ...                         ...
```

```
SELECT distributors.name
    FROM   distributors
    WHERE  distributors.name LIKE 'W%'
UNION
SELECT actors.name
    FROM   actors
    WHERE  actors.name LIKE 'W%';

      name
----------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

Compatibility

Extensions

PostgreSQL allows one to omit the `FROM` clause from a query. This feature was retained from the original PostQuel query language. It has a straightforward use to compute the results of simple constant expressions:

```
SELECT 2+2;

 ?column?
----------
        4
```

Some other DBMSes cannot do this except by introducing a dummy one-row table to do the select from. A less obvious use is to abbreviate a normal select from one or more tables:

```
SELECT distributors.* WHERE distributors.name = 'Westward';

 did | name
-----+----------
 108 | Westward
```

This works because an implicit FROM item is added for each table that is referenced in the query but not mentioned in FROM. While this is a convenient shorthand, it's easy to misuse. For example, the query

```
SELECT distributors.* FROM distributors d;
```

is probably a mistake; most likely the user meant

```
SELECT d.* FROM distributors d;
```

rather than the unconstrained join

```
SELECT distributors.* FROM distributors d, distributors distributors;
```

that he will actually get. To help detect this sort of mistake, PostgreSQL 7.1 and later will warn if the implicit-FROM feature is used in a query that also contains an explicit FROM clause.

SQL92

SELECT Clause

In the SQL92 standard, the optional keyword `AS` is just noise and can be omitted without affecting the meaning. The PostgreSQL parser requires this keyword when renaming output columns because the type extensibility features lead to parsing ambiguities in this context. `AS` is optional in FROM items, however.

The DISTINCT ON phrase is not part of SQL92. Nor are LIMIT and OFFSET.

In SQL92, an ORDER BY clause may only use result column names or numbers, while a GROUP BY clause may only use input column names. PostgreSQL extends each of these clauses to allow the other choice as well (but it uses the standard's interpretation if there is ambiguity). PostgreSQL also allows both clauses to specify arbitrary expressions. Note that names appearing in an expression will always be taken as input-column names, not as result-column names.

UNION/INTERSECT/EXCEPT Clause

The SQL92 syntax for UNION/INTERSECT/EXCEPT allows an additional CORRESPONDING BY option:

```
table_query UNION [ALL]
    [CORRESPONDING [BY (column [,...])]]
    table_query
```

The CORRESPONDING BY clause is not supported by PostgreSQL.

Name

SELECT INTO — create a new table from the results of a query

Synopsis

<refsynopsisdivinfo>2000-12-11</refsynopsisdivinfo>

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS output_name ] [, ...]
    INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ FOR UPDATE [ OF tablename [, ...] ] ]
    [ LIMIT [ start , ] { count | ALL } ]
    [ OFFSET start ]

where from_item can be:

[ ONLY ] table_name [ * ]
    [ [ AS ] alias [ ( column_alias_list ) ] ]
|
( select )
    [ AS ] alias [ ( column_alias_list ) ]
|
from_item [ NATURAL ] join_type from_item
    [ ON join_condition | USING ( join_column_list ) ]
```

Inputs

TEMPORARY
TEMP

If TEMPORARY or TEMP is specified, the output table is created only within this session, and is automatically dropped on session exit. Existing permanent tables with the same name are not visible (in this session) while the temporary table exists. Any indexes created on a temporary table are automatically temporary as well.

new_table

The name of the new table to be created. This table must not already exist. However, a temporary table can be created that has the same name as an existing permanent table.

All other inputs are described in detail for SELECT.

Outputs

Refer to CREATE TABLE and SELECT for a summary of possible output messages.

Description

`SELECT INTO` creates a new table and fills it with data computed by a query. The data is not returned to the client, as it is with a normal `SELECT`. The new table's columns have the names and data types associated with the output columns of the `SELECT`.

Note

CREATE TABLE AS is functionally equivalent to `SELECT INTO`. `CREATE TABLE AS` is the recommended syntax, since `SELECT INTO` is not standard. In fact, this form of `SELECT INTO` is not available in PL/pgSQL or ecpg , because they interpret the INTO clause differently.

Compatibility

SQL92 uses `SELECT ... INTO` to represent selecting values into scalar variables of a host program, rather than creating a new table. This indeed is the usage found in PL/pgSQL and ecpg . The PostgreSQL usage of `SELECT INTO` to represent table creation is historical. It's best to use `CREATE TABLE AS` for this purpose in new code. (`CREATE TABLE AS` isn't standard either, but it's less likely to cause confusion.)

Name

SET — change a run-time parameter

Synopsis

```
SET variable { TO | = } { value | 'value' | DEFAULT }
SET TIME ZONE { 'timezone' | LOCAL | DEFAULT }
```

Inputs

variable

A settable run-time parameter.

value

New value of parameter. DEFAULT can be used to specify resetting the parameter to its default value. Lists of strings are allowed, but more complex constructs may need to be single or double quoted.

Description

The SET command changes run-time configuration parameters. The following parameters can be altered:

CLIENT_ENCODING
NAMES

Sets the multibyte client encoding. The specified encoding must be supported by the backend.

This option is only available if PostgreSQL is build with multibyte support.

DATESTYLE

Choose the date/time representation style. Two separate settings are made: the default date/time output and the interpretation of ambiguous input.

The following are date/time output styles:

ISO

Use ISO 8601-style dates and times (YYYY-MM-DD HH:MM:SS). This is the default.

SQL

Use Oracle/Ingres-style dates and times. Note that this style has nothing to do with SQL (which mandates ISO 8601 style), the naming of this option is a historical accident.

PostgreSQL

Use traditional PostgreSQL format.

German

Use dd.mm.yyyy for numeric date representations.

The following two options determine both a substyle of the "SQL" and "PostgreSQL" output formats and the preferred interpretation of ambiguous date input.

European

Use `dd/mm/yyyy` for numeric date representations.

NonEuropean
US

Use `mm/dd/yyyy` for numeric date representations.

A value for `SET DATESTYLE` can be one from the first list (output styles), or one from the second list (substyles), or one from each separated by a comma.

Date format initialization may be done by:

Setting the `PGDATESTYLE` environment variable. If PGDATESTYLE is set in the frontend environment of a client based on libpq, libpq will automatically set DATESTYLE to the value of PGDATESTYLE during connection start-up.
Running postmaster using the option `-o -e` to set dates to the `European` convention.

The `DateStyle` option is really only intended for porting applications. To format your date/time values to choice, use the `to_char` family of functions.

SEED

Sets the internal seed for the random number generator.

value

The value for the seed to be used by the `random` function. Allowed values are floating-point numbers between 0 and 1, which are then multiplied by 2^{31}-1. This product will silently overflow if a number outside the range is used.

The seed can also be set by invoking the `setseed` SQL function:

```
SELECT setseed(value);
```

SERVER_ENCODING

Sets the multibyte server encoding.

This option is only available if PostgreSQL was built with multibyte support.

TIME ZONE
TIMEZONE

Sets the default time zone for your session. Arguments can be an SQL time interval constant, an integer or double precision constant, or a string representing a time zone supported by the host operating system.

The possible values for time zone depends on your operating system. For example, on Linux `/usr/share/zoneinfo` contains the database of time zones.

Here are some valid values for time zone:

'PST8PDT'

Set the time zone for California.

'Portugal'

Set the time zone for Portugal.

'Europe/Rome'

Set the time zone for Italy.

7

Set the time zone to 7 hours offset west from GMT (equivalent to PDT).

INTERVAL '08:00' HOUR TO MINUTE

Set the time zone to 8 hours offset west from GMT (equivalent to PST).

LOCAL
DEFAULT

Set the time zone to your local time zone (the one that your operating system defaults to).

If an invalid time zone is specified, the time zone becomes GMT (on most systems anyway).

If the `PGTZ` environment variable is set in the frontend environment of a client based on libpq, libpq will automatically set TIMEZONE to the value of `PGTZ` during connection start-up.

An extended list of other run-time parameters can be found in the *Administrator's Guide*.

Use SHOW to show the current setting of a parameters.

Diagnostics

`SET VARIABLE`

Message returned if successful.

`ERROR: not a valid option name: name`

The parameter you tried to set does not exist.

`ERROR: permission denied`

You must be a superuser to have access to certain settings.

`ERROR: name can only be set at start-up`

Some parameters are fixed once the server is started.

Examples

Set the style of date to traditional PostgreSQL with European conventions:

```
SET DATESTYLE TO PostgreSQL,European;
```

Set the time zone for Berkeley, California, using double quotes to preserve the uppercase attributes of the time zone specifier (note that the date/time format is ISO here):

```
SET TIME ZONE "PST8PDT";
SELECT CURRENT_TIMESTAMP AS today;

         today
-----------------------
 1998-03-31 07:41:21-08
```

Set the time zone for Italy (note the required single or double quotes to handle the special characters):

```
SET TIME ZONE 'Europe/Rome';
SELECT CURRENT_TIMESTAMP AS today;

         today
-----------------------
 1998-03-31 17:41:31+02
```

Compatibility

SQL92

The second syntax shown above (`SET TIME ZONE`) attempts to mimic SQL92. However, SQL allows only numeric time zone offsets. All other parameter settings as well as the first syntax shown above are a PostgreSQL extension.

Name

SET CONSTRAINTS — set the constraint mode of the current transaction

Synopsis

<refsynopsisdivinfo>2000-06-01</refsynopsisdivinfo>

```
SET CONSTRAINTS { ALL | constraint [, ...] } { DEFERRED | IMMEDIATE }
```

Description

`SET CONSTRAINTS` sets the behavior of constraint evaluation in the current transaction. In `IMMEDIATE` mode, constraints are checked at the end of each statement. In `DEFERRED` mode, constraints are not checked until transaction commit.

Upon creation, a constraint is always give one of three characteristics: `INITIALLY DEFERRED`, `INITIALLY IMMEDIATE DEFERRABLE`, or `INITIALLY IMMEDIATE NOT DEFERRABLE`. The third class is not affected by the `SET CONSTRAINTS` command.

Currently, only foreign key constraints are affected by this setting. Check and unique constraints are always effectively initially immediate not deferrable.

Compatibility

SQL92, SQL99

`SET CONSTRAINT` is defined in SQL92 and SQL99.

<docinfo>2001-04-21</docinfo>

Name

SET SESSION AUTHORIZATION — set the session user identifier and the current user identifier of the current session

Synopsis

```
SET SESSION AUTHORIZATION 'username'
```

Description

This command sets the session user identifier and the current user identifier of the current SQL-session context to be *username*.

The session user identifier is initially set to be the (possibly authenticated) user name provided by the client. The current user identifier is normally equal to the session user identifier, but may change temporarily in the context of "setuid" functions and similar mechanisms. The current user identifier is relevant for permission checking.

Execution of this command is only permitted if the initial session user (the *authenticated user*) had the superuser privilege. This permission is kept for the duration of a connection; for example, it is possible to temporarily become an unprivileged user and later switch back to become a superuser.

Examples

```
SELECT SESSION_USER, CURRENT_USER;
 current_user | session_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;
 current_user | session_user
--------------+--------------
 paul         | paul
```

Compatibility

SQL99

SQL99 allows some other expressions to appear in place of the literal *username* which are not important in practice. PostgreSQL allows identifier syntax ("username"), which SQL does not. SQL does not allow this command during a transaction; PostgreSQL does not make this restriction because there is no reason to. The privileges necessary to execute this command are left implementation-defined by the standard.

<docinfo>2000-11-24</docinfo>

Name

SET TRANSACTION — set the characteristics of the current transaction

Synopsis

```
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL
    { READ COMMITTED | SERIALIZABLE }
```

Description

This command sets the transaction isolation level. The `SET TRANSACTION` command sets the characteristics for the current SQL-transaction. It has no effect on any subsequent transactions. This command cannot be used after the first query or data-modification statement (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, `COPY`) of a transaction has been executed. `SET SESSION CHARACTERISTICS` sets the default transaction isolation level for each transaction for a session. `SET TRANSACTION` can override it for an individual transaction.

The isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently.

READ COMMITTED

A statement can only see rows committed before it began. This is the default.

SERIALIZABLE

The current transaction can only see rows committed before first query or data-modification statement was executed in this transaction.

Tip

Intuitively, serializable means that two concurrent transactions will leave the database in the same state as if the two has been executed strictly after one another in either order.

Notes

The session default transaction isolation level can also be set with the command

```
SET default_transaction_isolation = 'value'
```

and in the configuration file. Consult the *Administrator's Guide* for more information.

Compatibility

SQL92, SQL99

SERIALIZABLE is the default level in SQL. PostgreSQL does not provide the isolation levels `READ UNCOMMITTED` and `REPEATABLE READ`. Because of multiversion concurrency control, the serializable level is not truly serializable. See the *User's Guide* for details.

In SQL there are two other transaction characteristics that can be set with these commands: whether the transaction is read-only and the size of the diagnostics area. Neither of these concepts are supported in PostgreSQL.

Name

SHOW — show the value of a run-time parameter

Synopsis

```
SHOW name

SHOW ALL
```

Inputs

name

The name of a run-time parameter. See SET for a list.

ALL

Show all current session parameters.

Description

SHOW will display the current setting of a run-time parameter. These variables can be set using the SET statement or are determined at server start.

Diagnostics

ERROR: not a valid option name: *name*

Message returned if *variable* does not stand for an existing parameter.

ERROR: permission denied

You must be a superuser to be allowed to see certain settings.

NOTICE: Time zone is unknown

If the TZ or PGTZ environment variable is not set.

Examples

Show the current DateStyle setting:

```
SHOW DateStyle;
NOTICE:  DateStyle is ISO with US (NonEuropean) conventions
```

Show the current genetic optimizer (geqo) setting:

```
SHOW GEQO;
NOTICE:  geqo is on
```

Compatibility

The SHOW command is a PostgreSQL extension.

Name

TRUNCATE — empty a table

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
TRUNCATE [ TABLE ] name
```

Inputs

name

The name of the table to be truncated.

Outputs

`TRUNCATE`

Message returned if the table is successfully truncated.

Description

`TRUNCATE` quickly removes all rows from a table. It has the same effect as an unqualified `DELETE` but since it does not actually scan the table it is faster. This is most useful on large tables.

`TRUNCATE` cannot be executed inside a transaction block (`BEGIN`/`COMMIT` pair), because there is no way to roll it back.

Usage

Truncate the table `bigtable`:

```
TRUNCATE TABLE bigtable;
```

Compatibility

SQL92

There is no `TRUNCATE` in SQL92.

Name

UNLISTEN — stop listening for a notification

Synopsis

<refsynopsisdivinfo>1998-10-19</refsynopsisdivinfo>

```
UNLISTEN { notifyname | * }
```

Inputs

notifyname

Name of previously registered notify condition.

*

All current listen registrations for this backend are cleared.

Outputs

UNLISTEN

Acknowledgment that statement has executed.

Description

UNLISTEN is used to remove an existing NOTIFY registration. UNLISTEN cancels any existing registration of the current PostgreSQL session as a listener on the notify condition *notifyname*. The special condition wildcard * cancels all listener registrations for the current session.

NOTIFY contains a more extensive discussion of the use of LISTEN and NOTIFY.

Notes

notifyname need not be a valid class name but can be any string valid as a name up to 32 characters long.

The backend does not complain if you UNLISTEN something you were not listening for. Each backend will automatically execute UNLISTEN * when exiting.

Usage

To subscribe to an existing registration:

```
LISTEN virtual;
LISTEN
NOTIFY virtual;
NOTIFY
Asynchronous NOTIFY 'virtual' from backend with pid '8448' received
```

Once UNLISTEN has been executed, further NOTIFY commands will be ignored:

```
UNLISTEN virtual;
```

```
UNLISTEN
NOTIFY virtual;
NOTIFY
-- notice no NOTIFY event is received
```

Compatibility

SQL92

There is no `UNLISTEN` in SQL92.

Name

UPDATE — update rows of a table

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
UPDATE [ ONLY ] table SET col = expression [, ...]
    [ FROM fromlist ]
    [ WHERE condition ]
```

Inputs

table

The name of an existing table.

column

The name of a column in *table*.

expression

A valid expression or value to assign to column.

fromlist

A PostgreSQL non-standard extension to allow columns from other tables to appear in the WHERE condition.

condition

Refer to the SELECT statement for a further description of the WHERE clause.

Outputs

UPDATE #

Message returned if successful. The # means the number of rows updated. If # is 0 no rows are updated.

Description

UPDATE changes the values of the columns specified for all rows which satisfy condition. Only the columns to be modified need appear as columns in the statement.

Array references use the same syntax found in SELECT. That is, either single array elements, a range of array elements or the entire array may be replaced with a single query.

You must have write access to the table in order to modify it, as well as read access to any table whose values are mentioned in the WHERE condition.

By default UPDATE will update tuples in the table specified and all its sub-tables. If you wish to only update the specific table mentioned, you should use the ONLY clause.

Usage

Change word `Drama` with `Dramatic` on column `kind`:

```
UPDATE films
SET kind = 'Dramatic'
WHERE kind = 'Drama';
SELECT *
FROM films
WHERE kind = 'Dramatic' OR kind = 'Drama';
```

```
 code  |     title     | did | date_prod  |   kind   | len
-------+---------------+-----+------------+----------+-------
 BL101 | The Third Man | 101 | 1949-12-23 | Dramatic | 01:44
 P_302 | Becket        | 103 | 1964-02-03 | Dramatic | 02:28
 M_401 | War and Peace | 104 | 1967-02-12 | Dramatic | 05:57
 T_601 | Yojimbo       | 106 | 1961-06-16 | Dramatic | 01:50
 DA101 | Das Boot      | 110 | 1981-11-11 | Dramatic | 02:29
```

Compatibility

SQL92

SQL92 defines a different syntax for the positioned UPDATE statement:

```
UPDATE table SET column = expression [, ...]
    WHERE CURRENT OF cursor
```

where *`cursor`* identifies an open cursor.

Name

VACUUM — garbage-collect and optionally analyze a database

Synopsis

<refsynopsisdivinfo>2001-08-26</refsynopsisdivinfo>

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table [ (column
 [, ...] ) ] ]
```

Inputs

FULL

Selects “full” vacuum, which may reclaim more space, but takes much longer and exclusively locks the table.

FREEZE

Selects aggressive “freezing” of tuples.

VERBOSE

Prints a detailed vacuum activity report for each table.

ANALYZE

Updates statistics used by the optimizer to determine the most efficient way to execute a query.

table

The name of a specific table to vacuum. Defaults to all tables in the current database.

column

The name of a specific column to analyze. Defaults to all columns.

Outputs

VACUUM

The command is complete.

NOTICE: --Relation *table*--

The report header for *table*.

```
NOTICE: Pages 98: Changed 25, Reapped 74, Empty 0, New 0; Tup 1000: Vac
3000, Crash 0, UnUsed 0, MinLen 188, MaxLen 188; Re-using: Free/Avail.
Space 586952/586952; EndEmpty/Avail. Pages 0/74. Elapsed 0/0 sec.
```

The analysis for *table* itself.

```
NOTICE: Index index: Pages 28; Tuples 1000: Deleted 3000. Elapsed 0/0
sec.
```

The analysis for an index on the target table.

Description

VACUUM reclaims storage occupied by deleted tuples. In normal PostgreSQL operation, tuples that are DELETEd or obsoleted by UPDATE are not physically removed from their table; they remain present until a VACUUM is done. Therefore it's necessary to do VACUUM periodically, especially on frequently-updated tables.

With no parameter, VACUUM processes every table in the current database. With a parameter, VACUUM processes only that table.

VACUUM ANALYZE performs a VACUUM and then an ANALYZE for each selected table. This is a handy combination form for routine maintenance scripts. See ANALYZE for more details about its processing.

Plain VACUUM (without FULL) simply reclaims space and makes it available for re-use. This form of the command can operate in parallel with normal reading and writing of the table. VACUUM FULL does more extensive processing, including moving of tuples across blocks to try to compact the table to the minimum number of disk blocks. This form is much slower and requires an exclusive lock on each table while it is being processed.

FREEZE is a special-purpose option that causes tuples to be marked “frozen” as soon as possible, rather than waiting until they are quite old. If this is done when there are no other open transactions in the same database, then it is guaranteed that all tuples in the database are “frozen” and will not be subject to transaction ID wraparound problems, no matter how long the database is left un-vacuumed. FREEZE is not recommended for routine use. Its only intended usage is in connection with preparation of user-defined template databases, or other databases that are completely read-only and will not receive routine maintenance VACUUM operations. See the *Administrator's Guide* for details.

Notes

We recommend that active production databases be VACUUM-ed frequently (at least nightly), in order to remove expired rows. After adding or deleting a large number of records, it may be a good idea to issue a VACUUM ANALYZE command for the affected table. This will update the system catalogs with the results of all recent changes, and allow the PostgreSQL query optimizer to make better choices in planning user queries.

The FULL option is not recommended for routine use, but may be useful in special cases. An example is when you have deleted most of the rows in a table and would like the table to physically shrink to occupy less disk space. VACUUM FULL will usually shrink the table more than a plain VACUUM would.

Usage

The following is an example from running VACUUM on a table in the regression database:

```
regression=> VACUUM VERBOSE ANALYZE onek;
NOTICE:  --Relation onek--
NOTICE:  Index onek_unique1: Pages 14; Tuples 1000: Deleted 3000.
        CPU 0.00s/0.11u sec elapsed 0.12 sec.
NOTICE:  Index onek_unique2: Pages 16; Tuples 1000: Deleted 3000.
        CPU 0.00s/0.10u sec elapsed 0.10 sec.
NOTICE:  Index onek_hundred: Pages 13; Tuples 1000: Deleted 3000.
        CPU 0.00s/0.10u sec elapsed 0.10 sec.
NOTICE:  Index onek_stringu1: Pages 31; Tuples 1000: Deleted 3000.
        CPU 0.01s/0.09u sec elapsed 0.10 sec.
NOTICE:  Removed 3000 tuples in 70 pages.
```

```
        CPU 0.02s/0.04u sec elapsed 0.07 sec.
NOTICE:  Pages 94: Changed 0, Empty 0; Tup 1000: Vac 3000, Keep 0,
 UnUsed 0.
        Total CPU 0.05s/0.45u sec elapsed 0.59 sec.
NOTICE:  Analyzing onek
VACUUM
```

Compatibility

SQL92

There is no `VACUUM` statement in SQL92.

PostgreSQL Client Applications

This part contains reference information for PostgreSQL client applications and utilities. Not all of these commands are of general utility, some may require special privileges. The common feature of these applications is that they can be run on any host, independent of where the database server resides.

Table of Contents

createdb 510
createlang 512
createuser 514
dropdb 517
droplang 519
dropuser 521
ecpg 523
pgaccess 527
pg_config 529
pg_dump 531
pg_dumpall 537
pg_restore 539
psql 545
pgtclsh 566
pgtksh 567
vacuumdb 568

<docinfo>2000-11-11</docinfo>

Name

createdb — create a new PostgreSQL database

Synopsis

`createdb` [*options*...] [*dbname*] [*description*]

Inputs

-h, --host *host*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

-p, --port *port*

Specifies the Internet TCP/IP port or the local Unix domain socket file extension on which the server is listening for connections.

-U, --username *username*

User name to connect as

-W, --password

Force password prompt.

-e, --echo

Echo the queries that createdb generates and sends to the server.

-q, --quiet

Do not display a response.

-D, --location *datadir*

Specifies the alternative location for the database. See also initlocation.

-T, --template *template*

Specifies the template database from which to build this database.

-E, --encoding *encoding*

Specifies the character encoding scheme to be used in this database.

dbname

Specifies the name of the database to be created. The name must be unique among all PostgreSQL databases in this installation. The default is to create a database with the same name as the current system user.

description

This optionally specifies a comment to be associated with the newly created database.

The options `-h`, `-p`, `-U`, `-W`, and `-e` are passed on literally to psql. The options `-D`, `-T`, and `-E` are converted into options for the underlying SQL command CREATE DATABASE ; see there for more information about them.

Outputs

`CREATE DATABASE`

The database was successfully created.

`createdb: Database creation failed.`

(Says it all.)

`createdb: Comment creation failed. (Database was created.)`

The comment/description for the database could not be created. The database itself will have been created already. You can use the SQL command `COMMENT ON DATABASE` to create the comment later on.

If there is an error condition, the backend error message will be displayed. See CREATE DATABASE and psql for possibilities.

Description

createdb creates a new PostgreSQL database. The user who executes this command becomes the database owner.

createdb is a shell script wrapper around the SQL command CREATE DATABASE via the PostgreSQL interactive terminal psql. Thus, there is nothing special about creating databases via this or other methods. This means that the psql program must be found by the script and that a database server must be running at the targeted port. Also, any default settings and environment variables available to psql and the libpq front-end library will apply.

Usage

To create the database `demo` using the default database server:

```
$ createdb demo
CREATE DATABASE
```

The response is the same as you would have gotten from running the `CREATE DATABASE` SQL command.

To create the database `demo` using the server on host `eden`, port 5000, using the `LATIN1` encoding scheme with a look at the underlying query:

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE "demo" WITH ENCODING = 'LATIN1'
CREATE DATABASE
```

<docinfo>2001-05-09</docinfo>

Name

createlang — define a new PostgreSQL procedural language

Synopsis

```
createlang [connection-options...] langname [dbname]
createlang [connection-options...] [--list] | [-l] dbname
```

Inputs

createlang accepts the following command line arguments:

langname

Specifies the name of the procedural programming language to be defined.

-d, --dbname *dbname*

Specifies to which database the language should be added. The default is to use the database with the same name as the current system user.

-e, --echo

Displays SQL commands as they are executed.

-l, --list

Shows a list of already installed languages in the target database (which must be specified).

--L *directory*

Specifies the directory in which the language interpreter is to be found. The directory is normally found automatically; this option is primarily for debugging purposes.

createlang also accepts the following command line arguments for connection parameters:

-h, --host *host*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

-p, --port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections.

-U, --username *username*

User name to connect as

-W, --password

Force password prompt.

Outputs

Most error messages are self-explanatory. If not, run createlang with the `--echo` option and see under the respective SQL command for details. Check also under psql for more possibilities.

Description

createlang is a utility for adding a new programming language to a PostgreSQL database. createlang can handle all the languages supplied in the default PostgreSQL distribution, but not languages provided by other parties.

Although backend programming languages can be added directly using several SQL commands, it is recommended to use createlang because it performs a number of checks and is much easier to use. See CREATE LANGUAGE for more.

Notes

Use droplang to remove a language.

createlang is a shell script that invokes psql several times. If you have things arranged so that a password prompt is required to connect, you will be prompted for a password several times.

Usage

To install `pltcl` into the database `template1`:

```
$ createlang pltcl template1
```

<docinfo>2000-11-11</docinfo>

Name

createuser — define a new PostgreSQL user account

Synopsis

`createuser` [*options*...] [*username*]

Inputs

-h, --host *host*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

-p, --port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections.

-e, --echo

Echo the queries that createuser generates and sends to the server.

-q, --quiet

Do not display a response.

-d, --createdb

The new user is allowed to create databases.

-D, --no-createdb

The new user is not allowed to create databases.

-a, --adduser

The new user is allowed to create other users. (Note: actually, this makes the new user a *superuser*. The option is poorly named.)

-A, --no-adduser

The new user is not allowed to create other users (i.e., the new user is a regular user not a superuser).

-P, --pwprompt

If given, createuser will issue a prompt for the password of the new user. This is not necessary if you do not plan on using password authentication.

-i, --sysid *uid*

Allows you to pick a non-default user id for the new user. This is not necessary, but some people like it.

-E, --encrypted

Encrypts the user's password stored in the database. If not specified, the default is used.

-N, --unencrypted

Does not encrypt the user's password stored in the database. If not specified, the default is used.

username

Specifies the name of the PostgreSQL user to be created. This name must be unique among all PostgreSQL users.

You will be prompted for a name and other missing information if it is not specified on the command line.

The options `-h`, `-p`, and `-e`, are passed on literally to psql. The psql options `-U` and `-W` are available as well, but their use can be confusing in this context.

Outputs

`CREATE USER`

All is well.

`createuser: creation of user "`*username*`" failed`

Something went wrong. The user was not created.

If there is an error condition, the backend error message will be displayed. See CREATE USER and psql for possibilities.

Description

createuser creates a new PostgreSQL user. Only superusers (users with `usesuper` set in the `pg_shadow` table) can create new PostgreSQL users, so createuser must be invoked by someone who is a PostgreSQL superuser.

Being a superuser also implies the ability to bypass access permission checks within the database, so superuser-dom should not be granted lightly.

createuser is a shell script wrapper around the SQL command CREATE USER via the PostgreSQL interactive terminal psql. Thus, there is nothing special about creating users via this or other methods. This means that the psql application must be found by the script and that a database server must be running at the targeted host. Also, any default settings and environment variables used by psql and the libpq front-end library will apply.

Usage

To create a user `joe` on the default database server:

```
$ createuser joe
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
```

To create the same user `joe` using the server on host `eden`, port 5000, avoiding the prompts and taking a look at the underlying query:

```
$ createuser -p 5000 -h eden -D -A -e joe
```

```
CREATE USER "joe" NOCREATEDB NOCREATEUSER
CREATE USER
```

<docinfo>2000-11-11</docinfo>

Name

dropdb — remove a PostgreSQL database

Synopsis

`dropdb` [*options*...] *dbname*

Inputs

-h, --host *host*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

-p, --port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections.

-U, --username *username*

User name to connect as

-W, --password

Force password prompt.

-e, --echo

Echo the queries that dropdb generates and sends to the server.

-q, --quiet

Do not display a response.

-i, --interactive

Issues a verification prompt before doing anything destructive.

dbname

Specifies the name of the database to be removed. The database must be one of the existing PostgreSQL databases in this installation.

The options `-h`, `-p`, `-U`, `-W`, and `-e` are passed on literally to psql.

Outputs

`DROP DATABASE`

The database was successfully removed.

`dropdb: Database removal failed.`

Something didn't work out.

If there is an error condition, the backend error message will be displayed. See DROP DATABASE and psql for possibilities.

Description

dropdb destroys an existing PostgreSQL database. The user who executes this command must be a database superuser or the owner of the database.

dropdb is a shell script wrapper around the SQL command DROP DATABASE via the PostgreSQL interactive terminal psql. Thus, there is nothing special about dropping databases via this or other methods. This means that the psql must be found by the script and that a database server is running at the targeted host. Also, any default settings and environment variables available to psql and the libpq front-end library do apply.

Usage

To destroy the database `demo` on the default database server:

```
$ dropdb demo
DROP DATABASE
```

To destroy the database `demo` using the server on host `eden`, port 5000, with verification and a peek at the underlying query:

```
$ 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
```

<docinfo>2001-05-09</docinfo>

Name

droplang — remove a PostgreSQL procedural language

Synopsis

```
droplang [connection-options...] langname [dbname]
droplang [connection-options...] [--list] | [-l] dbname
```

Inputs

droplang accepts the following command line arguments:

langname

Specifies the name of the backend programming language to be removed.

[-d, --dbname] *dbname*

Specifies from which database the language should be removed. The default is to use the database with the same name as the current system user.

-e, --echo

Displays SQL commands as they are executed.

-l, --list

Shows a list of already installed languages in the target database (which must be specified).

droplang also accepts the following command line arguments for connection parameters:

-h, --host *host*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

-p, --port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections.

-U, --username *username*

User name to connect as

-W, --password

Force password prompt.

Outputs

Most error messages are self-explanatory. If not, run droplang with the `--echo` option and see under the respective SQL command for details. Check also under psql for more possibilities.

Description

droplang is a utility for removing an existing programming language from a PostgreSQL database. droplang can drop any procedural language, even those not supplied by the PostgreSQL distribution.

Although backend programming languages can be removed directly using several SQL commands, it is recommended to use droplang because it performs a number of checks and is much easier to use. See DROP LANGUAGE for more.

Notes

Use createlang to add a language.

Usage

To remove `pltcl`:

```
$ droplang pltcl dbname
```

<docinfo>2000-11-11</docinfo>

Name

dropuser — remove a PostgreSQL user account

Synopsis

`dropuser` [*options*...] [*username*]

Inputs

-h, --host *host*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

-p, --port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections.

-e, --echo

Echo the queries that createdb generates and sends to the server.

-q, --quiet

Do not display a response.

-i, --interactive

Prompt for confirmation before actually removing the user.

username

Specifies the name of the PostgreSQL user to be removed. This name must exist in the PostgreSQL installation. You will be prompted for a name if none is specified on the command line.

The options `-h`, `-p`, and `-e`, are passed on literally to psql. The psql options `-U` and `-W` are available as well, but they can be confusing in this context.

Outputs

`DROP USER`

All is well.

`dropuser: deletion of user "`*username*`" failed`

Something went wrong. The user was not removed.

If there is an error condition, the backend error message will be displayed. See DROP USER and psql for possibilities.

Description

dropuser removes an existing PostgreSQL user *and* the databases which that user owned. Only users with `usesuper` set in the `pg_shadow` table can destroy PostgreSQL users.

dropuser is a shell script wrapper around the SQL command DROP USER via the PostgreSQL interactive terminal psql. Thus, there is nothing special about removing users via this or other methods. This means that the psql must be found by the script and that a database server is running at the targeted host. Also, any default settings and environment variables available to psql and the libpq front-end library do apply.

Usage

To remove user `joe` from the default database server:

```
$ dropuser joe
DROP USER
```

To remove user `joe` using the postmaster on host `eden`, port 5000, with verification and a peek at the underlying query:

```
$ dropuser -p 5000 -h eden -i -e joe
User "joe" and any owned databases will be permanently deleted.
Are you sure? (y/n) y
DROP USER "joe"
DROP USER
```

Name

ecpg — embedded SQL C preprocessor

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

`ecpg` [-v] [-t] [-I *`include-path`*] [-o *`outfile`*] *`file`*...

Inputs

ecpg accepts the following command line arguments:

-v

Print version information.

-t

Turn on auto-commit of transactions. In this mode, each query is automatically committed unless it is inside an explicit transaction block. In the default mode, queries are committed only when `exec sql commit` is issued.

-I *`include-path`*

Specify an additional include path. Defaults are `.` (current directory), `/usr/local/include`, the PostgreSQL include path which is defined at compile time (default: `/usr/local/pgsql/include`), and `/usr/include`.

-o *`outfile`*

Specifies that ecpg should write all its output to *`outfile`*. If no such option is given the output is written to *`name`*`.c`, assuming the input file was named *`name`*`.pgc`. If the input file does have the expected `.pgc` suffix, then the output file will have `.pgc` appended to the input file name.

`file`

The files to be processed.

Outputs

ecpg will create a file or write to `stdout`.

Return value

ecpg returns 0 to the shell on successful completion, non-zero for errors.

Description

ecpg is an embedded SQL preprocessor for the C language and the PostgreSQL. It enables development of C programs with embedded SQL code.

Linus Tolke (<`linus@epact.se`>) was the original author of ecpg (up to version 0.2). Michael Meskes (<`meskes@debian.org`>) is the current author and maintainer of ecpg. Thomas Good

(<tomg@q8.nrnet.org>) is the author of the last revision of the ecpg man page, on which this document is based.

Usage

Preprocessing for Compilation

An embedded SQL source file must be preprocessed before compilation:

```
ecpg [ -d ] [ -o file ] file.pgc
```

where the optional `-d` flag turns on debugging. The `.pgc` extension is an arbitrary means of denoting ecpg source.

You may want to redirect the preprocessor output to a log file.

Compiling and Linking

Assuming the PostgreSQL binaries are in `/usr/local/pgsql`, you will need to compile and link your preprocessed source file:

```
gcc -g -I /usr/local/pgsql/include [ -o file ] file.c -L /usr/local/
pgsql/lib -lecpg -lpq
```

Grammar

Libraries

The preprocessor will prepend two directives to the source:

```
#include <ecpgtype.h>
#include <ecpglib.h>
```

Variable Declaration

Variables declared within ecpg source code must be prepended with:

```
EXEC SQL BEGIN DECLARE SECTION;
```

Similarly, variable declaration sections must terminate with:

```
EXEC SQL END DECLARE SECTION;
```

Note

Prior to version 2.1.0, each variable had to be declared on a separate line. As of version 2.1.0 multiple variables may be declared on a single line:

```
char  foo[16], bar[16];
```

Error Handling

The SQL communication area is defined with:

```
EXEC SQL INCLUDE sqlca;
```

Note

The `sqlca` is in lowercase. While SQL convention may be followed, i.e., using uppercase to separate embedded SQL from C statements, `sqlca` (which includes the `sqlca.h` header file) *must* be lowercase. This is because the EXEC SQL prefix indicates that this inclusion will be parsed by ecpg. ecpg observes case sensitivity (`SQLCA.h` will not be found). `EXEC SQL INCLUDE` can be used to include other header files as long as case sensitivity is observed.

The `sqlprint` command is used with the `EXEC SQL WHENEVER` statement to turn on error handling throughout the program:

```
EXEC SQL WHENEVER sqlerror sqlprint;
```

and

```
EXEC SQL WHENEVER not found sqlprint;
```

Note

This is *not* an exhaustive example of usage for the `EXEC SQL WHENEVER` statement. Further examples of usage may be found in SQL manuals (e.g., *The LAN TIMES Guide to SQL* by Groff and Weinberg).

Connecting to the Database Server

One connects to a database using the following:

```
EXEC SQL CONNECT TO dbname;
```

where the database name is not quoted. Prior to version 2.1.0, the database name was required to be inside single quotes.

Specifying a server and port name in the connect statement is also possible. The syntax is:

```
dbname[@server][:port]
```

or

```
<tcp|unix>:postgresql://server[:port][/dbname][?options]
```

Queries

In general, SQL queries acceptable to other applications such as psql can be embedded into your C code. Here are some examples of how to do that.

Create Table:

```
EXEC SQL CREATE TABLE foo (number int4, ascii char(16));
EXEC SQL CREATE UNIQUE index num1 on foo(number);
EXEC SQL COMMIT;
```

Insert:

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
```

```
EXEC SQL COMMIT;
```

Delete:

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

Singleton Select:

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

Select using Cursors:

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

Updates:

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```

Notes

The complete structure definition MUST be listed inside the declare section.

See the `TODO` file in the source for some more missing features.

<docinfo>2000-12-30</docinfo>

Name

pgaccess — a graphical PostgreSQL client application

Synopsis

`pgaccess` [*dbname*]

Options

dbname

The name of an existing database to access.

Description

PgAccess provides a graphical interface for PostgreSQL wherein you can manage your tables, edit them, define queries, sequences and functions.

PgAccess can:

- Open any database on a specified host at the specified port, user name, and password.
- Execute VACUUM .
- Save preferences in the `~/.pgaccessrc` file.

For tables, PgAccess can:

- Open multiple tables for viewing, with a configurable number of rows shown.
- Resize columns by dragging the vertical grid lines.
- Wrap text in cells.
- Dynamically adjust row height when editing.
- Save table layout for every table.
- Import/export to external files (SDF, CSV).
- Use filter capabilities; enter filters like `price > 3.14`.
- Specify sort order; enter manually the sort field(s).
- Edit in place; double click the text you want to change.
- Delete records; point to the record, press the **Delete** key.
- Add new records; save new row with right-button click.
- Create tables with an assistant.
- Rename and delete (drop) tables.
- Retrieve information on tables, including owner, field information, indexes.

For queries, PgAccess can:

- Define, edit and store user-defined queries.
- Save view layouts.
- Store queries as views.
- Execute with optional user input parameters, e.g.,

```
select * from invoices where year=[parameter "Year of selection"]
```

- View any select query result.
- Run action queries (insert, update, delete).
- Construct queries using a visual query builder with drag & drop support, table aliasing.

For sequences, PgAccess can:

- Define new instances.
- Inspect existing instances.
- Delete.

For views, PgAccess can:

- Define them by saving queries as views.
- View them, with filtering and sorting capabilities.
- Design new views.
- Delete (drop) existing views.

For functions, PgAccess can:

- Define.
- Inspect.
- Delete.

For reports, PgAccess can:

- Generate simple reports from a table (beta stage).
- Change font, size, and style of fields and labels.
- Load and save reports from the database.
- Preview tables, sample Postscript print.

For forms, PgAccess can:

- Open user-defined forms.
- Use a form design module.
- Access record sets using a query widget.

For scripts, PgAccess can:

- Define.
- Modify.
- Call user defined scripts.

Notes

PgAccess is written in Tcl/Tk. Your PostgreSQL installation needs to be built with Tcl support for PgAccess to be available.

<docinfo>2000-11-11</docinfo>

Name

pg_config — retrieve information about the installed version of PostgreSQL

Synopsis

`pg_config` {[--bindir] | [--includedir] | [--includedir-server] | [--libdir] | [--pkglibdir] | [--configure] | [--version]...}

Description

The pg_config utility prints configuration parameters of the currently installed version of PostgreSQL. It is intended, for example, to be used by software packages that want to interface to PostgreSQL to facilitate finding the required header files and libraries.

Options

To use pg_config, supply one or more of the following options:

`--bindir`

Print the location of user executables. Use this, for example, to find the psql program. This is normally also the location where the `pg_config` program resides.

`--includedir`

Print the location of C and C++ header files of the client interfaces.

`--includedir-server`

Print the location of C and C++ header files for server programming.

`--libdir`

Print the location of object code libraries.

`--pkglibdir`

Print the location of dynamically loadable modules, or where the server would search for them. (Other architecture-dependent data files may also be installed in this directory.)

`--configure`

Print the options that were given to the `configure` script when PostgreSQL was configured for building. This can be used to reproduce the identical configuration, or to find out with what options a binary package was built. (Note however that binary packages often contain vendor-specific custom patches.)

`--version`

Print the version of PostgreSQL and exit.

If more than one option (except for `--version`) is given, the information is printed in that order, one item per line.

Notes

The option `--includedir-server` is new in PostgreSQL 7.2. In prior releases, the server include files were installed in the same location as the client headers, which could be queried with the `--includedir`. To make your package handle both cases, try the newer option first and test the exit status to see whether it succeeded.

In releases prior to PostgreSQL 7.1, before the `pg_config` came to be, a method for finding the equivalent configuration information did not exist.

History

The `pg_config` utility first appeared in PostgreSQL 7.1.

See Also

PostgreSQL Programmer's Guide

<docinfo>2001-03-06</docinfo>

Name

pg_dump — extract a PostgreSQL database into a script file or other archive file

Synopsis

pg_dump [[-a] | [-s]] [-b] [-c] [-C] [[-d] | [-D]] [-f *file*] [-F *format*] [-i] [[-n] | [-N]] [-o] [-O] [-R] [-S] [-t *table*] [-v] [-x] [-X *keyword*] [-Z *0...9*] [-h *host*] [-p *port*] [-U *username*] [-W] *dbname*

Description

pg_dump is a utility for saving a PostgreSQL database into a script or an archive file. The script files are in plain-text format and contain the SQL commands required to reconstruct the database to the state it was in at the time it was saved. They can be used to reconstruct the database even on other machines and other architectures, with some modifications even on other RDBMS products. Furthermore, there are alternative archive file formats that are meant to be used with pg_restore to rebuild the database, and they also allow pg_restore to be selective about what is restored, or even to reorder the items prior to being restored. The archive files are also designed to be portable across architectures.

pg_dump will save the information necessary to re-generate all user-defined types, functions, tables, indexes, aggregates, and operators. In addition, all the data is copied out in text format so that it can be readily copied in again, as well as imported into tools for editing.

pg_dump is useful for dumping out the contents of a database to move from one PostgreSQL installation to another.

When used with one of the archive file formats and combined with pg_restore , pg_dump provides a flexible archival and transfer mechanism. pg_dump can be used to backup an entire database, then pg_restore can be used to examine the archive and/or select which parts of the database are to be restored. The most flexible output file format is the "custom" format (-Fc). It allows for selection and reordering of all archived items, and is compressed by default. The tar format (-Ft) is not compressed and it is not possible to reorder data when loading, but it is otherwise quite flexible; moreover, it can be manipulated with other tools such as tar.

While running pg_dump, one should examine the output for any warnings (printed on standard error), especially in light of the limitations listed below.

pg_dump makes consistent backups even if the database is being used concurrently. pg_dump does not block other users accessing the database (readers or writers).

Options

pg_dump accepts the following command line arguments. (Long option forms are only available on some platforms.)

dbname

Specifies the name of the database to be dumped.

-a
--data-only

Dump only the data, not the schema (data definitions).

This option is only meaningful for the plain-text format. For the other formats, you may specify the option when you call `pg_restore`.

-b
--blobs

Include large objects in dump.

-c
--clean

Output commands to clean (drop) database objects prior to (the commands for) creating them.

This option is only meaningful for the plain-text format. For the other formats, you may specify the option when you call `pg_restore`.

-C
--create

Begin the output with a command to create the database itself and reconnect to the created database. (With a script of this form, it doesn't matter which database you connect to before running the script.)

This option is only meaningful for the plain-text format. For the other formats, you may specify the option when you call `pg_restore`.

-d
--inserts

Dump data as `INSERT` commands (rather than `COPY`). This will make restoration very slow, but it makes the archives more portable to other RDBMS packages.

-D
--column-inserts
--attribute-inserts

Dump data as `INSERT` commands with explicit column names (`INSERT INTO` *`table`* `(`*`column`*`, ...) VALUES ...`). This will make restoration very slow, but it is necessary if you desire to rearrange column ordering.

-f *`file`*
--file=*`file`*

Send output to the specified file. If this is omitted, the standard output is used.

-F *`format`*
--format=*`format`*

Selects the format of the output. *`format`* can be one of the following:

p

Output a plain-text SQL script file (default)

t

Output a `tar` archive suitable for input into `pg_restore`. Using this archive format allows reordering and/or exclusion of schema elements at the time the database is restored. It is also possible to limit which data is reloaded at restore time.

c

Output a custom archive suitable for input into `pg_restore`. This is the most flexible format in that it allows reordering of data load as well as schema elements. This format is also compressed by default.

-i
--ignore-version

Ignore version mismatch between `pg_dump` and the database server. Since `pg_dump` knows a great deal about system catalogs, any given version of `pg_dump` is only intended to work with the corresponding release of the database server. Use this option if you need to override the version check (and if `pg_dump` then fails, don't say you weren't warned).

-n
--no-quotes

Suppress double quotes around identifiers unless absolutely necessary. This may cause trouble loading this dumped data if there are reserved words used for identifiers. This was the default behavior for `pg_dump` prior to version 6.4.

-N
--quotes

Include double quotes around identifiers. This is the default.

-o
--oids

Dump object identifiers (OIDs) for every table. Use this option if your application references the OID columns in some way (e.g., in a foreign key constraint). Otherwise, this option should not be used.

-O
--no-owner

Do not output commands to set the object ownership to match the original database. Typically, `pg_dump` issues (`psql`-specific) `\connect` statements to set ownership of schema elements. See also under `-R` and `-X use-set-session-authorization`. Note that `-O` does not prevent all reconnections to the database, only the ones that are exclusively used for ownership adjustments.

This option is only meaningful for the plain-text format. For the other formats, you may specify the option when you call `pg_restore`.

-R
--no-reconnect

Prohibit `pg_dump` from outputting a script that would require reconnections to the database while being restored. An average restoration script usually has to reconnect several times as different users to set the original ownerships of the objects. This option is a rather blunt instrument because it makes `pg_dump` lose this ownership information, *unless* you use the `-X use-set-session-authorization` option.

One possible reason why reconnections during restore might not be desired is if the access to the database requires manual interaction (e.g., passwords).

This option is only meaningful for the plain-text format. For the other formats, you may specify the option when you call `pg_restore`.

-s
--schema-only

Dump only the schema (data definitions), no data.

-S *username*
--superuser=*username*

The scripts or archives created by `pg_dump` need to have superuser access in certain cases, such as when disabling triggers or setting ownership of schema elements. This option specifies the user name to use for those cases.

-t *table*
--table=*table*

Dump data for *table* only.

-v
--verbose

Specifies verbose mode.

-x
--no-privileges
--no-acl

Prevent dumping of access privileges (grant/revoke commands).

-X use-set-session-authorization
--use-set-session-authorization

Normally, if a (plain-text mode) script generated by `pg_dump` must alter the current database user (e.g., to set correct object ownerships), it uses the psql `\connect` command. This command actually opens a new connection, which might require manual interaction (e.g., passwords). If you use the `-X use-set-session-authorization` option, then `pg_dump` will instead output SET SESSION AUTHORIZATION commands. This has the same effect, but it requires that the user restoring the database from the generated script be a database superuser. This option effectively overrides the `-R` option.

Since SET SESSION AUTHORIZATION is a standard SQL command, whereas `\connect` only works in psql, this option also enhances the theoretical portability of the output script.

This option is only meaningful for the plain-text format. For the other formats, you may specify the option when you call `pg_restore`.

-Z *0..9*
--compress=*0..9*

Specify the compression level to use in archive formats that support compression (currently only the custom archive format supports compression).

`pg_dump` also accepts the following command line arguments for connection parameters:

-h *host*
--host=*host*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

-p *port*
--port=*port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

-U *username*

Connect as the given user.

-W

Force a password prompt. This should happen automatically if the server requires password authentication.

Diagnostics

```
Connection to database 'template1' failed.
connectDBStart() -- connect() failed: No such file or directory
        Is the postmaster running locally
        and accepting connections on Unix socket '/tmp/.s.PGSQL.5432'?
```

pg_dump could not attach to the postmaster process on the specified host and port. If you see this message, ensure that the postmaster is running on the proper host and that you have specified the proper port.

Note

pg_dump internally executes SELECT statements. If you have problems running pg_dump, make sure you are able to select information from the database using, for example, psql.

Notes

If your installation has any local additions to the template1 database, be careful to restore the output of pg_dump into a truly empty database; otherwise you are likely to get errors due to duplicate definitions of the added objects. To make an empty database without any local additions, copy from template0 not template1, for example:

```
CREATE DATABASE foo WITH TEMPLATE = template0;
```

pg_dump has a few limitations:

- When dumping a single table or as plain text, pg_dump does not handle large objects. Large objects must be dumped in their entirety using one of the binary archive formats.
- When doing a data only dump, pg_dump emits queries to disable triggers on user tables before inserting the data and queries to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs may be left in the wrong state.

Examples

To dump a database:

```
$ pg_dump mydb > db.out
```

To reload this database:

```
$ psql -d database -f db.out
```

To dump a database called `mydb` that contains large objects to a `tar` file:

```
$ pg_dump -Ft -b mydb > db.tar
```

To reload this database (with large objects) to an existing database called `newdb`:

```
$ pg_restore -d newdb db.tar
```

History

The `pg_dump` utility first appeared in Postgres95 release `0.02`. The non-plain-text output formats were introduced in PostgreSQL release `7.1`.

See Also

pg_dumpall, pg_restore , psql, *PostgreSQL Administrator's Guide*

<docinfo>2000-12-19</docinfo>

Name

pg_dumpall — extract all PostgreSQL databases into a script file

Synopsis

`pg_dumpall` [[-c] | [--clean]] [[-g] | [--globals-only]] [-h *host*] [-p *port*] [-U *username*] [-W]

Description

pg_dumpall is a utility for writing out (“dumping”) all PostgreSQL databases of a cluster into one script file. The script file contains SQL commands that can be used as input to psql to restore the databases. It does this by calling pg_dump for each database in a cluster. pg_dumpall also dumps global objects that are common to all databases. (pg_dump does not save these objects.) This currently includes the information about database users and groups.

Thus, pg_dumpall is an integrated solution for backing up your databases. But note a limitation: it cannot dump “large objects”, since pg_dump cannot dump such objects into text files. If you have databases containing large objects, they should be dumped using one of pg_dump's non-text output modes.

Since pg_dumpall reads tables from all databases you will most likely have to connect as a database superuser in order to produce a complete dump. Also you will need superuser privileges to execute the saved script in order to be allowed to add users and groups, and to create databases.

The SQL script will be written to the standard output. Shell operators should be used to redirect it into a file.

Options

pg_dumpall accepts the following command line arguments:

-c, --clean

Include SQL commands to clean (drop) database objects before recreating them. (This option is fairly useless, since the output script expects to create the databases themselves; they would always be empty upon creation.)

-g, --globals-only

Only dump global objects (users and groups), no databases.

-h *host*

Specifies the host name of the machine on which the database server is running. If host begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

-p *port*

The port number on which the server is listening. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

-U *username*

Connect as the given user.

-W

Force a password prompt. This should happen automatically if the server requires password authentication.

Any other command line parameters are passed to the underlying pg_dump calls. This is useful to control some aspects of the output format, but some options such as `-f`, `-t`, and *`dbname`* should be avoided.

Examples

To dump all databases:

```
$ pg_dumpall > db.out
```

To reload this database use, for example:

```
$ psql -f db.out template1
```

(It is not important to which database you connect here since the script file created by pg_dumpall will contain the appropriate commands to create and connect to the saved databases.)

See Also

pg_dump , psql. Check there for details on possible error conditions.

<docinfo>2001-03-06</docinfo>

Name

pg_restore — restore a PostgreSQL database from an archive file created by pg_dump

Synopsis

pg_restore [-a] [-c] [-C] [-d *dbname*] [-f *output-file*] [-F *format*] [-i *index*] [-l] [-L *contents-file*] [[-N] | [-o] | [-r]] [-O] [-P *function-name*] [-R] [-s] [-S] [-t *table*] [-T *trigger*] [-v] [-x] [-X *keyword*] [-h *host*] [-p *port*] [-U *username*] [-W] [*archive-file*]

Description

pg_restore is a utility for restoring a PostgreSQL database from an archive created by pg_dump in one of the non-plain-text formats. It will issue the commands necessary to re-generate all user-defined types, functions, tables, indexes, aggregates, and operators, as well as the data in the tables.

The archive files contain information for pg_restore to rebuild the database, but also allow pg_restore to be selective about what is restored, or even to reorder the items prior to being restored. The archive files are designed to be portable across architectures.

pg_restore can operate in two modes: If a database name is specified, the archive is restored directly into the database. Otherwise, a script containing the SQL commands necessary to rebuild the database is created (and written to a file or standard output), similar to the ones created by the pg_dump plain text format. Some of the options controlling the script output are therefore analogous to pg_dump options.

Obviously, pg_restore cannot restore information that is not present in the archive file; for instance, if the archive was made using the "dump data as INSERTs" option, pg_restore will not be able to load the data using COPY statements.

Options

pg_restore accepts the following command line arguments. (Long option forms are only available on some platforms.)

archive-name

Specifies the location of the archive file to be restored. If not specified, the standard input is used.

-a
--data-only

Restore only the data, no schema (definitions).

-c
--clean

Clean (drop) database objects before recreating them.

-C
--create

Create the database before restoring into it. (When this switch appears, the database named with -d is used only to issue the initial CREATE DATABASE command. All data is restored into the database name that appears in the archive.)

-d *dbname*
--dbname=*dbname*

Connect to database *dbname* and restore directly into the database. Large objects can only be restored by using a direct database connection.

-f *filename*
--file=*filename*

Specify output file for generated script, or for the listing when used with `-l`. Default is the standard output.

-F *format*
--format=*format*

Specify format of the archive. It is not necessary to specify the format, since `pg_restore` will determine the format automatically. If specified, it can be one of the following:

t

Archive is a `tar` archive. Using this archive format allows reordering and/or exclusion of schema elements at the time the database is restored. It is also possible to limit which data is reloaded at restore time.

c

Archive is in the custom format of `pg_dump`. This is the most flexible format in that it allows reordering of data load as well as schema elements. This format is also compressed by default.

-i *index*
--index=*index*

Restore definition for named *index* only.

-l
--list

List the contents of the archive. The output of this command can be used with the `-L` option to restrict and reorder the items that are restored.

-L *list-file*
--use-list=*list-file*

Restore elements in *list-file* only, and in the order they appear in the file. Lines can be moved and may also be commented out by placing a `;` at the start of the line.

-N
--orig-order

Restore items in the original dump order. By default `pg_dump` will dump items in an order convenient to `pg_dump`, then save the archive in a modified OID order. This option overrides the OID ordering.

-o
--oid-order

Restore items in the OID order. By default `pg_dump` will dump items in an order convenient to `pg_dump`, then save the archive in a modified OID order. This option enforces strict OID ordering.

-O
--no-owner

Prevent any attempt to restore original object ownership. Objects will be owned by the user name used to attach to the database.

-P *function-name*
--function=*function-name*

Specify a procedure or function to be restored.

-r
--rearrange

Restore items in modified OID order. By default `pg_dump` will dump items in an order convenient to `pg_dump`, then save the archive in a modified OID order. Most objects will be restored in OID order, but some things (e.g., rules and indexes) will be restored at the end of the process irrespective of their OIDs. This option is the default.

-R
--no-reconnect

While restoring an archive, `pg_restore` typically has to reconnect to the database several times with different user names to set the correct ownership of the created objects. If this is undesirable (e.g., because manual interaction (passwords) would be necessary for each reconnection), this option prevents `pg_restore` from issuing any reconnection requests. (A connection request while in plain text mode, not connected to a database, is made by putting out a psql `\connect` command.) However, this option is a rather blunt instrument because it makes `pg_restore` lose all object ownership information, *unless* you use the `-X use-set-session-authorization` option.

-s
--schema-only

Restore the schema (definitions), no data. Sequence values will be reset.

-S *username*
--superuser=*username*

Specify the superuser user name to use when disabling triggers and/or setting ownership of schema elements. By default, `pg_restore` will use the current user name if it is a superuser.

-t *table*
--table=*table*

Restore schema/data for *table* only.

-T *trigger*
--trigger=*trigger*

Restore definition of *trigger* only.

-v
--verbose

Specifies verbose mode.

-x
--no-privileges
--no-acl

Prevent restoration of access privileges (grant/revoke commands).

-X use-set-session-authorization
--use-set-session-authorization

Normally, if restoring an archive requires altering the current database user (e.g., to set correct object ownerships), a new connection to the database must be opened, which might require manual interaction (e.g., passwords). If you use the `-X use-set-session-authorization` option, then `pg_restore` will instead use the SET SESSION AUTHORIZATION command. This has the same effect, but it requires that the user restoring the archive is a database superuser. This option effectively overrides the `-R` option.

`pg_restore` also accepts the following command line arguments for connection parameters:

-h *host*
--host=*host*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

-p *port*
--port=*port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections. The port number defaults to 5432, or the value of the `PGPORT` environment variable (if set).

-U *username*

Connect as the given user.

-W

Force a password prompt. This should happen automatically if the server requires password authentication.

Diagnostics

```
Connection to database 'template1' failed.
connectDBStart() -- connect() failed: No such file or directory
        Is the postmaster running locally
        and accepting connections on Unix socket '/tmp/.s.PGSQL.5432'?
```

`pg_restore` could not attach to the `postmaster` process on the specified host and port. If you see this message, ensure that the server is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

Note

When a direct database connection is specified using the -d option, `pg_restore` internally executes `SQL` statements. If you have problems running `pg_restore`, make sure you are able to select information from the database using, for example, `psql`.

Notes

If your installation has any local additions to the template1 database, be careful to load the output of pg_restore into a truly empty database; otherwise you are likely to get errors due to duplicate definitions of the added objects. To make an empty database without any local additions, copy from template0 not template1, for example:

```
CREATE DATABASE foo WITH TEMPLATE = template0;
```

The limitations of pg_restore are detailed below.

- When restoring data to a pre-existing table, pg_restore emits queries to disable triggers on user tables before inserting the data then emits queries to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs may be left in the wrong state.
- pg_restore will not restore large objects for a single table. If an archive contains large objects, then all large objects will be restored.

See the pg_dump documentation for details on limitations of pg_dump.

Examples

To dump a database:

```
$ pg_dump mydb > db.out
```

To reload this database:

```
$ psql -d database -f db.out
```

To dump a database called mydb that contains large objects to a tar file:

```
$ pg_dump -Ft -b mydb > db.tar
```

To reload this database (with large objects) to an existing database called newdb:

```
$ pg_restore -d newdb db.tar
```

To reorder database items, it is first necessary to dump the table of contents of the archive:

```
$ pg_restore -l archive.file > archive.list
```

The listing file consists of a header and one line for each item, e.g.,

```
;
; 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
```

Semi-colons are comment delimiters, and the numbers at the start of lines refer to the internal archive ID assigned to each item.

Lines in the file can be commented out, deleted, and reordered. For example,

```
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
```

could be used as input to `pg_restore` and would only restore items 10 and 6, in that order.

```
$ pg_restore -L archive.list archive.file
```

History

The `pg_restore` utility first appeared in PostgreSQL 7.1.

See Also

pg_dump , pg_dumpall, psql, *PostgreSQL Administrator's Guide*

<docinfo>2000-12-25</docinfo>

Name

psql — PostgreSQL interactive terminal

Synopsis

<refsynopsisdivinfo>1999-10-26</refsynopsisdivinfo>

```
psql [ options ] [ dbname [ user ] ]
```

Summary

psql is a terminal-based front-end to PostgreSQL. It enables you to type in queries interactively, issue them to PostgreSQL, and see the query results. Alternatively, input can be from a file. In addition, it provides a number of meta-commands and various shell-like features to facilitate writing scripts and automating a wide variety of tasks.

Description

Connecting To A Database

psql is a regular PostgreSQL client application. In order to connect to a database you need to know the name of your target database, the hostname and port number of the server and what user name you want to connect as. psql can be told about those parameters via command line options, namely `-d`, `-h`, `-p`, and `-U` respectively. If an argument is found that does not belong to any option it will be interpreted as the database name (or the user name, if the database name is also given). Not all these options are required, defaults do apply. If you omit the host name psql will connect via a Unix domain socket to a server on the local host. The default port number is compile-time determined. Since the database server uses the same default, you will not have to specify the port in most cases. The default user name is your Unix username, as is the default database name. Note that you can't just connect to any database under any username. Your database administrator should have informed you about your access rights. To save you some typing you can also set the environment variables `PGDATABASE`, `PGHOST`, `PGPORT` and `PGUSER` to appropriate values.

If the connection could not be made for any reason (e.g., insufficient privileges, postmaster is not running on the server, etc.), psql will return an error and terminate.

Entering Queries

In normal operation, psql provides a prompt with the name of the database to which psql is currently connected, followed by the string =>. For example,

```
$ psql testdb
Welcome to psql, 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

testdb=>
```

At the prompt, the user may type in SQL queries. Ordinarily, input lines are sent to the backend when a query-terminating semicolon is reached. An end of line does not terminate a query! Thus queries can be

spread over several lines for clarity. If the query was sent and without error, the query results are displayed on the screen.

Whenever a query is executed, psql also polls for asynchronous notification events generated by LISTEN and NOTIFY .

psql Meta-Commands

Anything you enter in psql that begins with an unquoted backslash is a psql meta-command that is processed by psql itself. These commands are what makes psql interesting for administration or scripting. Meta-commands are more commonly called slash or backslash commands.

The format of a psql command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of whitespace characters.

To include whitespace into an argument you must quote it with a single quote. To include a single quote into such an argument, precede it by a backslash. Anything contained in single quotes is furthermore subject to C-like substitutions for `\n` (new line), `\t` (tab), `\`*`digits`*, `\0`*`digits`*, and `\0x`*`digits`* (the character with the given decimal, octal, or hexadecimal code).

If an unquoted argument begins with a colon (`:`), it is taken as a variable and the value of the variable is taken as the argument instead.

Arguments that are quoted in "backticks" (`` ` ``) are taken as a command line that is passed to the shell. The output of the command (with a trailing newline removed) is taken as the argument value. The above escape sequences also apply in backticks.

Some commands take the name of an SQL identifier (such as a table name) as argument. These arguments follow the syntax rules of SQL regarding double quotes: an identifier without double quotes is coerced to lower-case. For all other commands double quotes are not special and will become part of the argument.

Parsing for arguments stops when another unquoted backslash occurs. This is taken as the beginning of a new meta-command. The special sequence `\\` (two backslashes) marks the end of arguments and continues parsing SQL queries, if any. That way SQL and psql commands can be freely mixed on a line. But in any case, the arguments of a meta-command cannot continue beyond the end of the line.

The following meta-commands are defined:

`\a`

If the current table output format is unaligned, switch to aligned. If it is not unaligned, set it to unaligned. This command is kept for backwards compatibility. See `\pset` for a general solution.

`\cd` [*`directory`*]

Change the current working directory to *`directory`*. Without argument, change to the current user's home directory.

Tip

To print your current working directory, use `\!pwd`.

`\C` [*`title`*]

Set the title of any tables being printed as the result of a query or unset any such title. This command is equivalent to `\pset title` *`title`*. (The name of this command derives from "caption", as it was previously only used to set the caption in an HTML table.)

`\connect` (or `\c`) [*dbname* [*username*]]

Establishes a connection to a new database and/or under a user name. The previous connection is closed. If *dbname* is - the current database name is assumed.

If *username* is omitted the current user name is assumed.

As a special rule, `\connect` without any arguments will connect to the default database as the default user (as you would have gotten by starting psql without any arguments).

If the connection attempt failed (wrong username, access denied, etc.), the previous connection will be kept if and only if psql is in interactive mode. When executing a non-interactive script, processing will immediately stop with an error. This distinction was chosen as a user convenience against typos on the one hand, and a safety mechanism that scripts are not accidentally acting on the wrong database on the other hand.

`\copy` *table* [`with oids`] { `from` | `to` } *filename* | stdin | stdout [`using delimiters` '*characters*'] [`with null as` '*string*']

Performs a frontend (client) copy. This is an operation that runs an SQL COPY command, but instead of the backend's reading or writing the specified file, and consequently requiring backend access and special user privilege, as well as being bound to the file system accessible by the backend, psql reads or writes the file and routes the data between the backend and the local file system.

The syntax of the command is similar to that of the SQL `COPY` command (see its description for the details). Note that, because of this, special parsing rules apply to the `\copy` command. In particular, the variable substitution rules and backslash escapes do not apply.

Tip

This operation is not as efficient as the SQL `COPY` command because all data must pass through the client/server IP or socket connection. For large amounts of data the other technique may be preferable.

Note

Note the difference in interpretation of `stdin` and `stdout` between frontend and backend copies: in a frontend copy these always refer to psql's input and output stream. On a backend copy `stdin` comes from wherever the `COPY` itself came from (for example, a script run with the `-f` option), and `stdout` refers to the query output stream (see `\o` meta-command below).

`\copyright`

Shows the copyright and distribution terms of PostgreSQL.

`\d` *relation*

Shows all columns of *relation* (which could be a table, view, index, or sequence), their types, and any special attributes such as `NOT NULL` or defaults, if any. If the relation is, in fact, a table, any defined indices, primary keys, unique constraints and check constraints are also listed. If the relation is a view, the view definition is also shown.

The command form `\d+` is identical, but any comments associated with the table columns are shown as well.

Note

If \d is called without any arguments, it is equivalent to \dtvs which will show a list of all tables, views, and sequences. This is purely a convenience measure.

\da [*pattern*]

Lists all available aggregate functions, together with the data type they operate on. If *pattern* (a regular expression) is specified, only matching aggregates are shown.

\dd [*object*]

Shows the descriptions of *object* (which can be a regular expression), or of all objects if no argument is given. ("Object" covers aggregates, functions, operators, types, relations (tables, views, indexes, sequences, large objects), rules, and triggers.) For example:

```
=> \dd version
              Object descriptions
  Name   |   What   |         Description
---------+----------+-------------------------
 version | function | PostgreSQL version string
(1 row)
```

Descriptions for objects can be generated with the COMMENT ON SQL command.

Note

PostgreSQL stores the object descriptions in the pg_description system table.

\df [*pattern*]

Lists available functions, together with their argument and return types. If *pattern* (a regular expression) is specified, only matching functions are shown. If the form \df+ is used, additional information about each function, including language and description, is shown.

\distvS [*pattern*]

This is not the actual command name: The letters i, s, t, v, S stand for index, sequence, table, view, and system table, respectively. You can specify any or all of them in any order to obtain a listing of them, together with who the owner is.

If *pattern* is specified, it is a regular expression that restricts the listing to those objects whose name matches. If one appends a "+" to the command name, each object is listed with its associated description, if any.

\dl

This is an alias for \lo_list, which shows a list of large objects.

\do [*name*]

Lists available operators with their operand and return types. If *name* is specified, only operators with that name will be shown.

\dp [*pattern*]

This is an alias for \z which was included for its greater mnemonic value ("display permissions").

`\dT [ pattern ]`

Lists all data types or only those that match *pattern*. The command form `\dT+` shows extra information.

`\du [ pattern ]`

Lists all configured users or only those that match *pattern*.

`\edit` (or `\e`) [*filename*]

If *filename* is specified, the file is edited; after the editor exits, its content is copied back to the query buffer. If no argument is given, the current query buffer is copied to a temporary file which is then edited in the same fashion.

The new query buffer is then re-parsed according to the normal rules of psql, where the whole buffer is treated as a single line. (Thus you cannot make scripts this way. Use `\i` for that.) This means also that if the query ends with (or rather contains) a semicolon, it is immediately executed. In other cases it will merely wait in the query buffer.

Tip

psql searches the environment variables `PSQL_EDITOR`, `EDITOR`, and `VISUAL` (in that order) for an editor to use. If all of them are unset, `/bin/vi` is run.

`\echo` *text* [...]

Prints the arguments to the standard output, separated by one space and followed by a newline. This can be useful to intersperse information in the output of scripts. For example:

```
=> \echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

If the first argument is an unquoted `-n` the the trailing newline is not written.

Tip

If you use the `\o` command to redirect your query output you may wish to use `\qecho` instead of this command.

`\encoding` [*encoding*]

Sets the client encoding, if you are using multibyte encodings. Without an argument, this command shows the current encoding.

`\f` [*string*]

Sets the field separator for unaligned query output. The default is pipe (`|`). See also `\pset` for a generic way of setting output options.

`\g` [{ *filename* | `|` *command* }]

Sends the current query input buffer to the backend and optionally saves the output in *filename* or pipes the output into a separate Unix shell to execute *command*. A bare `\g` is virtually equivalent to a semicolon. A `\g` with argument is a "one-shot" alternative to the `\o` command.

`\help` (or `\h`) [*command*]

Give syntax help on the specified SQL command. If *command* is not specified, then psql will list all the commands for which syntax help is available. If *command* is an asterisk ("*"), then syntax help on all SQL commands is shown.

Note

To simplify typing, commands that consists of several words do not have to be quoted. Thus it is fine to type **\help alter table**.

`\H`

Turns on HTML query output format. If the HTML format is already on, it is switched back to the default aligned text format. This command is for compatibility and convenience, but see `\pset` about setting other output options.

`\i` *filename*

Reads input from the file *filename* and executes it as though it had been typed on the keyboard.

Note

If you want to see the lines on the screen as they are read you must set the variable `ECHO` to `all`.

`\l` (or `\list`)

List all the databases in the server as well as their owners. Append a "+" to the command name to see any descriptions for the databases as well. If your PostgreSQL installation was compiled with multibyte encoding support, the encoding scheme of each database is shown as well.

`\lo_export` *loid filename*

Reads the large object with OID *loid* from the database and writes it to *filename*. Note that this is subtly different from the server function `lo_export`, which acts with the permissions of the user that the database server runs as and on the server's file system.

Tip

Use `\lo_list` to find out the large object's OID.

Note

See the description of the `LO_TRANSACTION` variable for important information concerning all large object operations.

`\lo_import` *filename* [*comment*]

Stores the file into a PostgreSQL "large object". Optionally, it associates the given comment with the object. Example:

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

The response indicates that the large object received object id 152801 which one ought to remember if one wants to access the object ever again. For that reason it is recommended to always associate a human-readable comment with every object. Those can then be seen with the `\lo_list` command.

Note that this command is subtly different from the server-side `lo_import` because it acts as the local user on the local file system, rather than the server's user and file system.

Note

See the description of the `LO_TRANSACTION` variable for important information concerning all large object operations.

`\lo_list`

Shows a list of all PostgreSQL "large objects" currently stored in the database, along with any comments provided for them.

`\lo_unlink` *`loid`*

Deletes the large object with OID *`loid`* from the database.

Tip

Use `\lo_list` to find out the large object's OID.

Note

See the description of the `LO_TRANSACTION` variable for important information concerning all large object operations.

`\o` [{*`filename`* | `|` *`command`*}]

Saves future query results to the file *`filename`* or pipes future results into a separate Unix shell to execute *`command`*. If no arguments are specified, the query output will be reset to `stdout`.

"Query results" includes all tables, command responses, and notices obtained from the database server, as well as output of various backslash commands that query the database (such as `\d`), but not error messages.

Tip

To intersperse text output in between query results, use `\qecho`.

`\p`

Print the current query buffer to the standard output.

`\pset` *`parameter`* [*`value`*]

This command sets options affecting the output of query result tables. *`parameter`* describes which option is to be set. The semantics of *`value`* depend thereon.

Adjustable printing options are:

`format`

Sets the output format to one of `unaligned`, `aligned`, `html`, or `latex`. Unique abbreviations are allowed. (That would mean one letter is enough.)

"Unaligned" writes all fields of a tuple on a line, separated by the currently active field separator. This is intended to create output that might be intended to be read in by other programs (tab-separated, comma-separated). "Aligned" mode is the standard, human-readable, nicely formatted text output that is default. The "HTML" and "LaTeX" modes put out tables that are intended to be included in documents using the respective mark-up language. They are not complete documents! (This might not be so dramatic in HTML, but in LaTeX you must have a complete document wrapper.)

`border`

The second argument must be a number. In general, the higher the number the more borders and lines the tables will have, but this depends on the particular format. In HTML mode, this will translate directly into the `border=...` attribute, in the others only values 0 (no border), 1 (internal dividing lines), and 2 (table frame) make sense.

`expanded` (or `x`)

Toggles between regular and expanded format. When expanded format is enabled, all output has two columns with the field name on the left and the data on the right. This mode is useful if the data wouldn't fit on the screen in the normal "horizontal" mode.

Expanded mode is supported by all four output modes.

`null`

The second argument is a string that should be printed whenever a field is null. The default is not to print anything, which can easily be mistaken for, say, an empty string. Thus, one might choose to write `\pset null '(null)'`.

`fieldsep`

Specifies the field separator to be used in unaligned output mode. That way one can create, for example, tab- or comma-separated output, which other programs might prefer. To set a tab as field separator, type `\pset fieldsep '\t'`. The default field separator is `'|'` (a "pipe" symbol).

`footer`

Toggles the display of the default footer `(x rows)`.

`recordsep`

Specifies the record (line) separator to use in unaligned output mode. The default is a newline character.

`tuples_only` (or `t`)

Toggles between tuples only and full display. Full display may show extra information such as column headers, titles, and various footers. In tuples only mode, only actual table data is shown.

`title` [*text*]

Sets the table title for any subsequently printed tables. This can be used to give your output descriptive tags. If no argument is given, the title is unset.

Note

This formerly only affected HTML mode. You can now set titles in any output format.

`tableattr` (or `T`) [*text*]

Allows you to specify any attributes to be placed inside the HTML `table` tag. This could for example be `cellpadding` or `bgcolor`. Note that you probably don't want to specify `border` here, as that is already taken care of by `\pset border`.

`pager`

Toggles the list of a pager to do table output. If the environment variable `PAGER` is set, the output is piped to the specified program. Otherwise `more` is used.

In any case, psql only uses the pager if it seems appropriate. That means among other things that the output is to a terminal and that the table would normally not fit on the screen. Because of the modular nature of the printing routines it is not always possible to predict the number of lines that will actually be printed. For that reason psql might not appear very discriminating about when to use the pager and when not to.

Illustrations on how these different formats look can be seen in the Examples section.

Tip

There are various shortcut commands for `\pset`. See `\a`, `\C`, `\H`, `\t`, `\T`, and `\x`.

Note

It is an error to call `\pset` without arguments. In the future this call might show the current status of all printing options.

`\q`

Quit the psql program.

`\qecho` *text* [...]

This command is identical to `\echo` except that all output will be written to the query output channel, as set by `\o`.

`\r`

Resets (clears) the query buffer.

`\s` [*filename*]

Print or save the command line history to *filename*. If *filename* is omitted, the history is written to the standard output. This option is only available if psql is configured to use the GNU history library.

Note

In the current version, it is no longer necessary to save the command history, since that will be done automatically on program termination. The history is also loaded automatically every time psql starts up.

`\set` [*name* [*value* [...]]]

Sets the internal variable *name* to *value* or, if more than one value is given, to the concatenation of all of them. If no second argument is given, the variable is just set with no value. To unset a variable, use the `\unset` command.

Valid variable names can contain characters, digits, and underscores. See the section about psql variables for details.

Although you are welcome to set any variable to anything you want, psql treats several variables as special. They are documented in the section about variables.

Note

This command is totally separate from the SQL command SET.

`\t`

Toggles the display of output column name headings and row count footer. This command is equivalent to `\pset tuples_only` and is provided for convenience.

`\T` *table_options*

Allows you to specify options to be placed within the `table` tag in HTML tabular output mode. This command is equivalent to `\pset tableattr` *table_options*.

`\w` {*filename*|`|`*command*}

Outputs the current query buffer to the file *filename* or pipes it to the Unix command *command*.

`\x`

Toggles extended row format mode. As such it is equivalent to `\pset expanded`.

`\z` [*pattern*]

Produces a list of all tables in the database with their appropriate access permissions listed. If an argument is given it is taken as a regular expression which limits the listing to those tables which match it.

```
test=> \z
Access permissions for database "test"
 Relation |           Access permissions
----------+-------------------------------------
 my_table | {"=r","joe=arwR", "group staff=ar"}
(1 row )
```

Read this as follows:

- `"=r"`: `PUBLIC` has read (`SELECT`) permission on the table.
- `"joe=arwR"`: User `joe` has read, write (`UPDATE`, `DELETE`), “append” (`INSERT`) permissions, and permission to create rules on the table.
- `"group staff=ar"`: Group `staff` has `SELECT` and `INSERT` permission.

The commands GRANT and REVOKE are used to set access permissions.

`\!` [*command*]

Escapes to a separate Unix shell or executes the Unix command *command*. The arguments are not further interpreted, the shell will see them as is.

`\?`

Get help information about the backslash (“\”) commands.

Command-line Options

If so configured, psql understands both standard Unix short options, and GNU-style long options. The latter are not available on all systems.

-a, --echo-all

Print all the lines to the screen as they are read. This is more useful for script processing rather than interactive mode. This is equivalent to setting the variable `ECHO` to `all`.

-A, --no-align

Switches to unaligned output mode. (The default output mode is otherwise aligned.)

-c, --command *`query`*

Specifies that psql is to execute one query string, *`query`*, and then exit. This is useful in shell scripts.

`query` must be either a query string that is completely parseable by the backend (i.e., it contains no psql specific features), or it is a single backslash command. Thus you cannot mix SQL and psql meta-commands. To achieve that, you could pipe the string into psql, like this: `echo "\x \\ select * from foo;" | psql`.

-d, --dbname *`dbname`*

Specifies the name of the database to connect to. This is equivalent to specifying *`dbname`* as the first non-option argument on the command line.

-e, --echo-queries

Show all queries that are sent to the backend. This is equivalent to setting the variable `ECHO` to `queries`.

-E, --echo-hidden

Echoes the actual queries generated by \d and other backslash commands. You can use this if you wish to include similar functionality into your own programs. This is equivalent to setting the variable `ECHO_HIDDEN` from within psql.

-f, --file *`filename`*

Use the file *`filename`* as the source of queries instead of reading queries interactively. After the file is processed, psql terminates. This is in many ways equivalent to the internal command `\i`.

If *`filename`* is - (hyphen), then standard input is read.

Using this option is subtly different from writing `psql < `*`filename`*. In general, both will do what you expect, but using `-f` enables some nice features such as error messages with line numbers. There is also a slight chance that using this option will reduce the start-up overhead. On the other hand, the variant using the shell's input redirection is (in theory) guaranteed to yield exactly the same output that you would have gotten had you entered everything by hand.

-F, --field-separator *`separator`*

Use *`separator`* as the field separator. This is equivalent to `\pset fieldsep` or `\f`.

-h, --host *hostname*

Specifies the host name of the machine on which the postmaster is running. If host begins with a slash, it is used as the directory for the unix domain socket.

-H, --html

Turns on HTML tabular output. This is equivalent to `\pset format html` or the `\H` command.

-l, --list

Lists all available databases, then exits. Other non-connection options are ignored. This is similar to the internal command `\list`.

-o, --output *filename*

Put all query output into file *filename*. This is equivalent to the command `\o`.

-p, --port *port*

Specifies the TCP/IP port or, by omission, the local Unix domain socket file extension on which the postmaster is listening for connections. Defaults to the value of the `PGPORT` environment variable or, if not set, to the port specified at compile time, usually 5432.

-P, --pset *assignment*

Allows you to specify printing options in the style of `\pset` on the command line. Note that here you have to separate name and value with an equal sign instead of a space. Thus to set the output format to LaTeX, you could write `-P format=latex`.

-q

Specifies that psql should do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this happens. This is useful with the `-c` option. Within psql you can also set the `QUIET` variable to achieve the same effect.

-R, --record-separator *separator*

Use *separator* as the record separator. This is equivalent to the `\pset recordsep` command.

-s, --single-step

Run in single-step mode. That means the user is prompted before each query is sent to the backend, with the option to cancel execution as well. Use this to debug scripts.

-S, --single-line

Runs in single-line mode where a newline terminates a query, as a semicolon does.

Note

This mode is provided for those who insist on it, but you are not necessarily encouraged to use it. In particular, if you mix SQL and meta-commands on a line the order of execution might not always be clear to the inexperienced user.

-t, --tuples-only

Turn off printing of column names and result row count footers, etc. It is completely equivalent to the `\t` meta-command.

-T, --table-attr *table_options*

Allows you to specify options to be placed within the HTML `table` tag. See `\pset` for details.

-u

Makes psql prompt for the user name and password before connecting to the database.

This option is deprecated, as it is conceptually flawed. (Prompting for a non-default user name and prompting for a password because the backend requires it are really two different things.) You are encouraged to look at the `-U` and `-W` options instead.

-U, --username *username*

Connects to the database as the user *username* instead of the default. (You must have permission to do so, of course.)

-v, --variable, --set *assignment*

Performs a variable assignment, like the `\set` internal command. Note that you must separate name and value, if any, by an equal sign on the command line. To unset a variable, leave off the equal sign. To just set a variable without a value, use the equal sign but leave off the value. These assignments are done during a very early stage of start-up, so variables reserved for internal purposes might get overwritten later.

-V, --version

Shows the psql version.

-W, --password

Requests that psql should prompt for a password before connecting to a database. This will remain set for the entire session, even if you change the database connection with the meta-command `\connect`.

In the current version, psql automatically issues a password prompt whenever the backend requests password authentication. Because this is currently based on a hack, the automatic recognition might mysteriously fail, hence this option to force a prompt. If no password prompt is issued and the backend requires password authentication the connection attempt will fail.

-x, --expanded

Turns on extended row format mode. This is equivalent to the command `\x`.

-X, --no-psqlrc

Do not read the start-up file `~/.psqlrc`.

-?, --help

Shows help about psql command line arguments.

Advanced features

Variables

psql provides variable substitution features similar to common Unix command shells. This feature is new and not very sophisticated, yet, but there are plans to expand it in the future. Variables are simply name/

value pairs, where the value can be any string of any length. To set variables, use the psql meta-command `\set`:

```
testdb=> \set foo bar
```

sets the variable "foo" to the value "bar". To retrieve the content of the variable, precede the name with a colon and use it as the argument of any slash command:

```
testdb=> \echo :foo
bar
```

Note

The arguments of `\set` are subject to the same substitution rules as with other commands. Thus you can construct interesting references such as `\set :foo 'something'` and get "soft links" or "variable variables" of Perl or PHP fame, respectively. Unfortunately (or fortunately?), there is no way to do anything useful with these constructs. On the other hand, `\set bar :foo` is a perfectly valid way to copy a variable.

If you call `\set` without a second argument, the variable is simply set, but has no value. To unset (or delete) a variable, use the command `\unset`.

psql's internal variable names can consist of letters, numbers, and underscores in any order and any number of them. A number of regular variables are treated specially by psql. They indicate certain option settings that can be changed at runtime by altering the value of the variable or represent some state of the application. Although you can use these variables for any other purpose, this is not recommended, as the program behavior might grow really strange really quickly. By convention, all specially treated variables consist of all upper-case letters (and possibly numbers and underscores). To ensure maximum compatibility in the future, avoid such variables. A list of all specially treated variables follows.

`DBNAME`

The name of the database you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

`ECHO`

If set to "`all`", all lines entered or from a script are written to the standard output before they are parsed or executed. To specify this on program start-up, use the switch `-a`. If set to "`queries`", psql merely prints all queries as they are sent to the backend. The option for this is `-e`.

`ECHO_HIDDEN`

When this variable is set and a backslash command queries the database, the query is first shown. This way you can study the PostgreSQL internals and provide similar functionality in your own programs. If you set the variable to the value "noexec", the queries are just shown but are not actually sent to the backend and executed.

`ENCODING`

The current client multibyte encoding. If you are not set up to use multibyte characters, this variable will always contain "SQL_ASCII".

`HISTCONTROL`

If this variable is set to `ignorespace`, lines which begin with a space are not entered into the history list. If set to a value of `ignoredups`, lines matching the previous history line are not entered. A

value of `ignoreboth` combines the two options. If unset, or if set to any other value than those above, all lines read in interactive mode are saved on the history list.

Note

This feature was shamelessly plagiarized from bash.

`HISTSIZE`

The number of commands to store in the command history. The default value is 500.

Note

This feature was shamelessly plagiarized from bash.

`HOST`

The database server host you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

`IGNOREEOF`

If unset, sending an EOF character (usually Control-D) to an interactive session of psql will terminate the application. If set to a numeric value, that many EOF characters are ignored before the application terminates. If the variable is set but has no numeric value, the default is 10.

Note

This feature was shamelessly plagiarized from bash.

`LASTOID`

The value of the last affected oid, as returned from an `INSERT` or `lo_insert` command. This variable is only guaranteed to be valid until after the result of the next SQL command has been displayed.

`LO_TRANSACTION`

If you use the PostgreSQL large object interface to specially store data that does not fit into one tuple, all the operations must be contained in a transaction block. (See the documentation of the large object interface for more information.) Since psql has no way to tell if you already have a transaction in progress when you call one of its internal commands (`\lo_export`, `\lo_import`, `\lo_unlink`) it must take some arbitrary action. This action could either be to roll back any transaction that might already be in progress, or to commit any such transaction, or to do nothing at all. In the last case you must provide your own `BEGIN TRANSACTION`/`COMMIT` block or the results will be unpredictable (usually resulting in the desired action's not being performed in any case).

To choose what you want to do you set this variable to one of “rollback”, “commit”, or “nothing”. The default is to roll back the transaction. If you just want to load one or a few objects this is fine. However, if you intend to transfer many large objects, it might be advisable to provide one explicit transaction block around all commands.

`ON_ERROR_STOP`

By default, if non-interactive scripts encounter an error, such as a malformed SQL query or internal meta-command, processing continues. This has been the traditional behavior of psql but it is

sometimes not desirable. If this variable is set, script processing will immediately terminate. If the script was called from another script it will terminate in the same fashion. If the outermost script was not called from an interactive psql session but rather using the `-f` option, psql will return error code 3, to distinguish this case from fatal error conditions (error code 1).

`PORT`

The database server port to which you are currently connected. This is set every time you connect to a database (including program start-up), but can be unset.

`PROMPT1`, `PROMPT2`, `PROMPT3`

These specify what the prompt psql issues is supposed to look like. See "Prompting" below.

`QUIET`

This variable is equivalent to the command line option `-q`. It is probably not too useful in interactive mode.

`SINGLELINE`

This variable is set by the command line option `-S`. You can unset or reset it at run time.

`SINGLESTEP`

This variable is equivalent to the command line option `-s`.

`USER`

The database user you are currently connected as. This is set every time you connect to a database (including program start-up), but can be unset.

SQL Interpolation

An additional useful feature of psql variables is that you can substitute ("interpolate") them into regular SQL statements. The syntax for this is again to prepend the variable name with a colon (`:`).

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

would then query the table `my_table`. The value of the variable is copied literally, so it can even contain unbalanced quotes or backslash commands. You must make sure that it makes sense where you put it. Variable interpolation will not be performed into quoted SQL entities.

A popular application of this facility is to refer to the last inserted OID in subsequent statements to build a foreign key scenario. Another possible use of this mechanism is to copy the contents of a file into a field. First load the file into a variable and then proceed as above.

```
testdb=> \set content '\'' `cat my_file.txt` '\''
testdb=> INSERT INTO my_table VALUES (:content);
```

One possible problem with this approach is that `my_file.txt` might contain single quotes. These need to be escaped so that they don't cause a syntax error when the third line is processed. This could be done with the program sed:

```
testdb=> \set content '\'' `sed -e "s/'/\\\\\\'/g" < my_file.txt` '\''
```

Observe the correct number of backslashes (6)! You can resolve it this way: After psql has parsed this line, it passes `sed -e "s/'/\\\'/g" < my_file.txt` to the shell. The shell will do its own thing inside the double quotes and execute `sed` with the arguments `-e` and `s/'/\\'/g`. When sed parses this it will replace the two backslashes with a single one and then do the substitution. Perhaps at one point you thought it was great that all Unix commands use the same escape character. And this is ignoring the fact that you might have to escape all backslashes as well because SQL text constants are also subject to certain interpretations. In that case you might be better off preparing the file externally.

Since colons may legally appear in queries, the following rule applies: If the variable is not set, the character sequence "colon+name" is not changed. In any case you can escape a colon with a backslash to protect it from interpretation. (The colon syntax for variables is standard SQL for embedded query languages, such as ecpg. The colon syntax for array slices and type casts are PostgreSQL extensions, hence the conflict.)

Prompting

The prompts psql issues can be customized to your preference. The three variables `PROMPT1`, `PROMPT2`, and `PROMPT3` contain strings and special escape sequences that describe the appearance of the prompt. Prompt 1 is the normal prompt that is issued when psql requests a new query. Prompt 2 is issued when more input is expected during query input because the query was not terminated with a semicolon or a quote was not closed. Prompt 3 is issued when you run an SQL `COPY` command and you are expected to type in the tuples on the terminal.

The value of the respective prompt variable is printed literally, except where a percent sign ("%") is encountered. Depending on the next character, certain other text is substituted instead. Defined substitutions are:

`%M`

The full hostname (with domain name) of the database server, or `[local]` if the connection is over a Unix domain socket, or `[local:`*`/dir/name`*`]`, if the Unix domain socket is not at the compiled in default location.

`%m`

The hostname of the database server, truncated after the first dot, or `[local]` if the connection is over a Unix domain socket.

`%>`

The port number at which the database server is listening.

`%n`

The username you are connected as (not your local system user name).

`%/`

The name of the current database.

`%~`

Like `%/`, but the output is "~" (tilde) if the database is your default database.

`%#`

If the current user is a database superuser, then a "#", otherwise a ">".

`%R`

In prompt 1 normally "=", but "^" if in single-line mode, and "!" if the session is disconnected from the database (which can happen if `\connect` fails). In prompt 2 the sequence is replaced by "-", "*", a single quote, or a double quote, depending on whether psql expects more input because the query wasn't terminated yet, because you are inside a `/* ... */` comment, or because you are inside a quote. In prompt 3 the sequence doesn't resolve to anything.

`%`*`digits`*

If *`digits`* starts with `0x` the rest of the characters are interpreted as a hexadecimal digit and the character with the corresponding code is substituted. If the first digit is `0` the characters are interpreted as on octal number and the corresponding character is substituted. Otherwise a decimal number is assumed.

`%:`*`name`*`:`

The value of the psql, variable *`name`*. See the section "Variables" for details.

`%` `` ` ``*`command`*`` ` ``

The output of *`command`*, similar to ordinary "back-tick" substitution.

To insert a percent sign into your prompt, write `%%`. The default prompts are equivalent to `'%/%R%# '` for prompts 1 and 2, and `'>> '` for prompt 3.

Note

This feature was shamelessly plagiarized from tcsh.

Miscellaneous

psql returns 0 to the shell if it finished normally, 1 if a fatal error of its own (out of memory, file not found) occurs, 2 if the connection to the backend went bad and the session is not interactive, and 3 if an error occurred in a script and the variable `ON_ERROR_STOP` was set.

Before starting up, psql attempts to read and execute commands from the file `$HOME/.psqlrc`. It could be used to set up the client or the server to taste (using the `\set` and `SET` commands).

GNU readline

psql supports the readline and history libraries for convenient line editing and retrieval. The command history is stored in a file named `.psql_history` in your home directory and is reloaded when psql starts up. Tab-completion is also supported, although the completion logic makes no claim to be an SQL parser. When available, psql is automatically built to use these features. If for some reason you do not like the tab completion, you can turn if off by putting this in a file named `.inputrc` in your home directory:

```
$if psql
set disable-completion on
$endif
```

(This is not a psql but a readline feature. Read its documentation for further details.)

If you have the readline library installed but psql does not seem to use it, you must make sure that PostgreSQL's top-level `configure` script finds it. `configure` needs to find both the library `libreadline.a` (or a shared library equivalent) *and* the header files `readline.h` and `history.h`

(or `readline/readline.h` and `readline/history.h`) in appropriate directories. If you have the library and header files installed in an obscure place you must tell `configure` about them, for example:

```
$ ./configure --with-includes=/opt/gnu/include --with-libs=/opt/gnu/
lib  ...
```

Then you have to recompile psql (not necessarily the entire code tree).

The GNU readline library can be obtained from the GNU project's FTP server at ftp://ftp.gnu.org.

Examples

Note

This section only shows a few examples specific to psql. If you want to learn SQL or get familiar with PostgreSQL, you might wish to read the Tutorial that is included in the distribution.

The first example shows how to spread a query over several lines of input. Notice the changing prompt:

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text
testdb-> );
CREATE
```

Now look at the table definition again:

```
testdb=> \d my_table
             Table "my_table"
 Attribute |  Type   |      Modifier
-----------+---------+--------------------
 first     | integer | not null default 0
 second    | text    |
```

At this point you decide to change the prompt to something more interesting:

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

Let's assume you have filled the table with data and want to take a look at it:

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

You can make this table look differently by using the `\pset` command:

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
```

```
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep ","
Field separator is ",".
peter@localhost testdb=> \pset tuples_only
Showing only tuples.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4
```

Alternatively, use the short commands:

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

Appendix

Bugs and Issues

- In some earlier life psql allowed the first argument to start directly after the (single-letter) command. For compatibility this is still supported to some extent but I am not going to explain the details here as this use is discouraged. But if you get strange messages, keep this in mind. For example

  ```
  testdb=> \foo
  Field separator is "oo",
  ```

  which is perhaps not what one would expect.

- psql only works smoothly with servers of the same version. That does not mean other combinations will fail outright, but subtle and not-so-subtle problems might come up.

- Pressing Control-C during a “copy in” (data sent to the server) doesn't show the most ideal of behaviors. If you get a message such as “COPY state must be terminated first”, simply reset the connection by entering `\c - -`.

<docinfo>2001-03-05</docinfo>

Name

pgtclsh — PostgreSQL Tcl shell client

Synopsis

`pgtclsh` [*`filename`* [*`arguments`*...]]

Description

`pgtclsh` is a Tcl shell interface extended with PostgreSQL database access functions. (Essentially, it is `tclsh` with `libpgtcl` loaded.) Like with the regular Tcl shell, the first command line argument is a script file, any remaining arguments are passed to the script. If no script file is named, the shell is interactive.

A Tcl shell with Tk and PostgreSQL functions is available as pgtksh .

See Also

pgtksh , *PostgreSQL Programmer's Guide* (description of `libpgtcl`) , tclsh

<docinfo>2001-03-05</docinfo>

Name

pgtksh — PostgreSQL Tcl/Tk shell client

Synopsis

`pgtksh` [*`filename`* [*`arguments`*...]]

Description

`pgtksh` is a Tcl/Tk shell interface extended with PostgreSQL database access functions. (Essentially, it is `wish` with `libpgtcl` loaded.) Like with `wish`, the regular Tcl/Tk shell, the first command line argument is a script file, any remaining arguments are passed to the script. Special options may be processed by the X Window System libraries instead. If no script file is named, the shell is interactive.

A plain Tcl shell with PostgreSQL functions is available as pgtclsh .

See Also

pgtclsh , *PostgreSQL Programmer's Guide* (description of `libpgtcl`) , tclsh , wish

<docinfo>2000-11-11</docinfo>

Name

vacuumdb — garbage-collect and analyze a PostgreSQL database

Synopsis

`vacuumdb` [*connection-options*...] [[-d] *dbname*] [[--full] | [-f]] [[--verbose] | [-v]] [[--analyze] | [-z]] [--table '*table* [(*column* [,...])]']
`vacuumdb` [*connection-options*...] [[--all] | [-a]] [[--full] | [-f]] [[--verbose] | [-v]] [[--analyze] | [-z]]

Inputs

vacuumdb accepts the following command line arguments:

-d *dbname*
--dbname *dbname*

Specifies the name of the database to be cleaned or analyzed.

-a
--all

Vacuum all databases.

-f
--full

Perform “full” vacuuming.

-v
--verbose

Print detailed information during processing.

-z
--analyze

Calculate statistics for use by the optimizer.

-t *table* [(*column* [,...])]
--table *table* [(*column* [,...])]

Clean or analyze *table* only. Column names may be specified only in conjunction with the `--analyze` option.

Tip

If you specify columns to vacuum, you probably have to escape the parentheses from the shell.

vacuumdb also accepts the following command line arguments for connection parameters:

-h *host*
--host *host*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

-p *port*
--port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections.

-U *username*
--username *username*

User name to connect as

-W
--password

Force password prompt.

-e
--echo

Echo the commands that vacuumdb generates and sends to the server.

-q
--quiet

Do not display a response.

Outputs

`VACUUM`

Everything went well.

`vacuumdb: Vacuum failed.`

Something went wrong. vacuumdb is only a wrapper script. See VACUUM and psql for a detailed discussion of error messages and potential problems.

Description

vacuumdb is a utility for cleaning a PostgreSQL database. vacuumdb will also generate internal statistics used by the PostgreSQL query optimizer.

vacuumdb is a shell script wrapper around the backend command VACUUM via the PostgreSQL interactive terminal psql. There is no effective difference between vacuuming databases via this or other methods. psql must be found by the script and a database server must be running at the targeted host. Also, any default settings and environment variables available to psql and the libpq front-end library do apply.

Usage

To clean the database `test`:

```
$ vacuumdb test
```

To clean and analyze for the optimizer a database named `bigdb`:

```
$ vacuumdb --analyze bigdb
```

To clean a single table `foo` in a database named `xyzzy`, and analyze a single column `bar` of the table for the optimizer:

```
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzy
```

PostgreSQL Server Applications

This part contains reference information for PostgreSQL server applications and support utilities. These commands can only be run usefully on the host where the database server resides. Other utility programs are listed in PostgreSQL Client Applications.

Table of Contents

initdb 572
initlocation 574
ipcclean 575
pg_ctl 576
pg_passwd 579
postgres 580
postmaster 583

<docinfo>2000-12-25</docinfo>

Name

initdb — create a new PostgreSQL database cluster

Synopsis

`initdb` [--pgdata] | [-D]*`directory`* [[--username] | [-U]*`username`*] [[--pwprompt] | [-W]] [[--encoding] | [-E]*`encoding`*] [-L *`directory`*] [[--noclean] | [-n]] [[--debug] | [-d]]

Description

`initdb` creates a new PostgreSQL database cluster (or database system). A database cluster is a collection of databases that are managed by a single server instance.

Creating a database system consists of creating the directories in which the database data will live, generating the shared catalog tables (tables that belong to the whole cluster rather than to any particular database), and creating the `template1` database. When you create a new database, everything in the `template1` database is copied. It contains catalog tables filled in for things like the built-in types.

`initdb` must be run as the user that will own the server process, because the server needs to have access to the files and directories that `initdb` creates. Since the server may not be run as root, you must not run `initdb` as root either. (It will in fact refuse to do so.)

Although `initdb` will attempt to create the specified data directory, often it won't have permission to do so, since the parent of the desired data directory is often a root-owned directory. To set up an arrangement like this, create an empty data directory as root, then use `chown` to hand over ownership of that directory to the database user account, then `su` to become the database user, and finally run `initdb` as the database user.

Options

--pgdata=*`directory`*
-D *`directory`*

This option specifies the directory where the database system should be stored. This is the only information required by `initdb`, but you can avoid writing it by setting the `PGDATA` environment variable, which can be convenient since the database server (`postmaster`) can find the database directory later by the same variable.

--username=*`username`*
-U *`username`*

Selects the user name of the database superuser. This defaults to the name of the effective user running `initdb`. It is really not important what the superuser's name is, but one might choose to keep the customary name “postgres”, even if the operating system user's name is different.

--pwprompt
-W

Makes `initdb` prompt for a password to give the database superuser. If you don't plan on using password authentication, this is not important. Otherwise you won't be able to use password authentication until you have a password set up.

--encoding=*encoding*
-E *encoding*

Selects the encoding of the template database. This will also be the default encoding of any database you create later, unless you override it there. To use the encoding feature, you must have enabled it at build time, at which time you also select the default for this option.

Other, less commonly used, parameters are also available:

-L *directory*

Specifies where `initdb` should find its input files to initialize the database system. This is normally not necessary. You will be told if you need to specify their location explicitly.

--noclean
-n

By default, when `initdb` determines that an error prevented it from completely creating the database system, it removes any files it may have created before discovering that it can't finish the job. This option inhibits tidying-up and is thus useful for debugging.

--debug
-d

Print debugging output from the bootstrap backend and a few other messages of lesser interest for the general public. The bootstrap backend is the program `initdb` uses to create the catalog tables. This option generates a tremendous amount of extremely boring output.

Environment

`PGDATA`

Specifies the directory where the database system is to be stored; may be overridden using the `-D` option.

See Also

postgres, postmaster, *PostgreSQL Administrator's Guide*

<docinfo>2000-11-11</docinfo>

Name

initlocation — create a secondary PostgreSQL database storage area

Synopsis

`initlocation` *`directory`*

Description

initlocation creates a new PostgreSQL secondary database storage area. See the discussion under CREATE DATABASE about how to manage and use secondary storage areas. If the argument does not contain a slash and is not valid as a path, it is assumed to be an environment variable, which is referenced. See the examples at the end.

In order to use this command you must be logged in (using `su`, for example) as the database superuser.

Usage

To create a database in an alternate location, using an environment variable:

```
$ export PGDATA2=/opt/postgres/data
```

Stop and start postmaster so it sees the `PGDATA2` environment variable. The system must be configured so the postmaster sees `PGDATA2` every time it starts. Finally:

```
$ initlocation PGDATA2
$ createdb -D PGDATA2 testdb
```

Alternatively, if you allow absolute paths you could write:

```
$ initlocation /opt/postgres/data
$ createdb -D /opt/postgres/data/testdb testdb
```

<docinfo>2000-11-11</docinfo>

Name

ipcclean — remove shared memory and semaphores from an aborted PostgreSQL server

Synopsis

```
ipcclean
```

Description

`ipcclean` removes all shared memory segments and semaphore sets owned by the current user. It is intended to be used for cleaning up after a crashed PostgreSQL server (postmaster). Note that immediately restarting the server will also clean up shared memory and semaphores, so this command is of little real utility.

Only the database administrator should execute this program as it can cause bizarre behavior (i.e., crashes) if run during multiuser execution. If this command is executed while a postmaster is running, the shared memory and semaphores allocated by the postmaster will be deleted. This will result in a general failure of the backend servers started by that postmaster.

Notes

This script is a hack, but in the many years since it was written, no one has come up with an equally effective and portable solution. Since the postmaster can now clean up by itself, it is unlikely that `ipcclean` will be improved upon in the future.

The script makes assumption about the format of output of the ipcs utility which may not be true across different operating systems. Therefore, it may not work on your particular OS.

<docinfo>2001-02-08</docinfo>

Name

pg_ctl — start, stop, or restart a PostgreSQL server

Synopsis

`pg_ctl` start [-w] [-s] [-D *`datadir`*] [-l *`filename`*] [-o *`options`*] [-p *`path`*]
`pg_ctl` stop [-W] [-s] [-D *`datadir`*] [-m [s[mart]] | [f[ast]] | [i[mmediate]]]
`pg_ctl` restart [-w] [-s] [-D *`datadir`*] [-m [s[mart]] | [f[ast]] | [i[mmediate]]] [-o *`options`*]
`pg_ctl` reload [-s] [-D *`datadir`*]
`pg_ctl` status [-D *`datadir`*]

Description

pg_ctl is a utility for starting, stopping, or restarting postmaster, the PostgreSQL backend server, or displaying the status of a running postmaster. Although the postmaster can be started manually, pg_ctl encapulates tasks such as redirecting log output, properly detaching from the terminal and process group, and it provides convenient options for controlled shutdown.

In `start` mode, a new postmaster is launched. The server is started in the background, the standard input attached to `/dev/null`. The standard output and standard error are either appended to a log file, if the `-l` option is used, or are redirected to pg_ctl's standard output (not standard error). If no log file is chosen, the standard output of pg_ctl should be redirected to a file or piped to another process, for example a log rotating program, otherwise the postmaster will write its output the the controlling terminal (from the background) and will not leave the shell's process group.

In `stop` mode, the postmaster that is running in the specified data directory is shut down. Three different shutdown methods can be selected with the `-m` option: “Smart” mode waits for all the clients to disconnect. This is the default. “Fast” mode does not wait for clients to disconnect. All active transactions are rolled back and clients are forcibly disconnected, then the database is shut down. “Immediate” mode will abort all server processes without clean shutdown. This will lead to a recovery run on restart.

`restart` mode effectively executes a stop followed by a start. This allows the changing of postmaster command line options.

`reload` mode simply sends the postmaster a SIGHUP signal, causing it to reread its configuration files (`postgresql.conf`, `pg_hba.conf`, etc.). This allows changing of configuration-file options that do not require a complete restart to take effect.

`status` mode checks whether a postmaster is running and if so displays the PID and the command line options that were used to invoke it.

Options

-D *`datadir`*

Specifies the file system location of the database files. If this is omitted, the environment variable `PGDATA` is used.

-l *`filename`*

Append the server log output to *`filename`*. If the file does not exist, it is created. The umask is set to 077, so access to the log file from other users is disallowed by default.

-m *mode*

Specifies the shutdown mode. *mode* may be `smart`, `fast`, or `immediate`, or the first letter of one of these three.

-o *options*

Specifies options to be passed directly to postmaster.

The parameters are usually surrounded by single or double quotes to ensure that they are passed through as a group.

-p *path*

Specifies the location of the `postmaster` executable. By default the postmaster is taken from the same directory as pg_ctl, or failing that, the hard-wired installation directory. It is not necessary to use this option unless you are doing something unusual and get errors that the postmaster was not found.

-s

Only print errors, no informational messages.

-w

Wait for the start or shutdown to complete. Times out after 60 seconds. This is the default for shutdowns.

-W

Do not wait for start or shutdown to complete. This is the default for starts and restarts.

Files

If the file `postmaster.opts.default` exists in the data directory, the contents of the file will be passed as options to the postmaster, unless overridden by the `-o` option.

Examples

Starting the postmaster

To start up a postmaster:

```
$ pg_ctl start
```

An example of starting the postmaster, blocking until the postmaster comes up is:

```
$ pg_ctl -w start
```

For a postmaster using port 5433, and running without `fsync`, use:

```
$ pg_ctl -o "-F -p 5433" start
```

Stopping the postmaster

```
$ pg_ctl stop
```

stops the postmaster. Using the `-m` switch allows one to control *how* the backend shuts down.

Restarting the postmaster

This is almost equivalent to stopping the postmaster and starting it again except that pg_ctl saves and reuses the command line options that were passed to the previously running instance. To restart the postmaster in the simplest form:

```
$ pg_ctl restart
```

To restart postmaster, waiting for it to shut down and to come up:

```
$ pg_ctl -w restart
```

To restart using port 5433 and disabling fsync after restarting:

```
$ pg_ctl -o "-F -p 5433" restart
```

Showing postmaster status

Here is a sample status output from 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'
```

This is the command line that would be invoked in restart mode.

Bugs

Waiting for complete start is not a well-defined operation and may fail if access control is set up so that a local client cannot connect without manual interaction. It should be avoided.

See Also

postmaster, *PostgreSQL Administrator's Guide*

<docinfo>2000-11-18</docinfo>

Name

pg_passwd — change a secondary PostgreSQL password file

Synopsis

```
pg_passwd filename
```

Description

pg_passwd is a tool for manipulating flat text password files. These files can control client authentication of the PostgreSQL server. More information about setting up this authentication mechanism can be found in the *Administrator's Guide*.

The format of a text password file is one entry per line; the fields of each entry are separated by colons. The first field is the user name, the second field is the encrypted password. Other fields are ignored (to allow password files to be shared between applications that use similar formats). pg_passwd enables users to interactively add entries to such a file, to alter passwords of existing entries, and to encrypt such passwords.

Supply the name of the password file as argument to the pg_passwd command. To be used by PostgreSQL, the file needs to be located in the server's data directory, and the base name of the file needs to be specified in the `pg_hba.conf` access control file.

```
$ pg_passwd /usr/local/pgsql/data/passwords
File "/usr/local/pgsql/data/passwords" does not exist.  Create? (y/
n): y
Username: guest
Password:
Re-enter password:
```

where the `Password:` and `Re-enter password:` prompts require the same password input which is not displayed on the terminal. Note that the password is limited to eight useful characters by restrictions of the standard crypt(3) library routine.

The original password file is renamed to `passwords.bk`.

To make use of this password file, put a line like the following in `pg_hba.conf`:

```
host  mydb     133.65.96.250   255.255.255.255 password passwords
```

which would allow access to database mydb from host 133.65.96.250 using the passwords listed in the `passwords` file (and only to the users listed in that file).

Note

It is also useful to have entries in a password file with empty password fields. (This is different from an empty password.) Such entries allow you to restrict users who can access the system. These entries cannot be managed by pg_passwd, but you can edit password files manually.

See also

PostgreSQL Administrator's Guide

<docinfo>2000-11-12</docinfo>

Name

postgres — run a PostgreSQL server in single-user mode

Synopsis

`postgres` [-A [0] | [1]] [-B *nbuffers*] [-c *name*=*value*] [-d *debug-level*] [-D *datadir*] [-e] [-E] [-f [s] | [i] | [t] | [n] | [m] | [h]] [-F] [-i] [-N] [-o *filename*] [-O] [-P] [[-s] | [-t [pa] | [pl] | [ex]]] [-S *sort-mem*] [-W *seconds*] [--*name*=*value*] *database*
`postgres` [-A [0] | [1]] [-B *nbuffers*] [-c *name*=*value*] [-d *debug-level*] [-D *datadir*] [-e] [-f [s] | [i] | [t] | [n] | [m] | [h]] [-F] [-i] [-o *filename*] [-O] [-p *database*] [-P] [[-s] | [-t [pa] | [pl] | [ex]]] [-S *sort-mem*] [-v *protocol-version*] [-W *seconds*] [--*name*=*value*]

Description

The `postgres` executable is the actual PostgreSQL server process that processes queries. It is normally not called directly; instead a postmaster multi-user server is started.

The second form above is how postgres is invoked by the postmaster (only conceptually, since both `postmaster` and `postgres` are in fact the same program); it should not be invoked directly this way. The first form invokes the server directly in interactive single-user mode. The primary use for this mode is during bootstrapping by initdb. Sometimes it is used for debugging or disaster recovery.

When invoked in interactive mode from the shell, the user can enter queries and the results will be printed to the screen, but in a form that is more useful for developers than end users. But note that running a single-user backend is not truly suitable for debugging the server since no realistic interprocess communication and locking will happen.

When running a stand-alone backend, the session user will be set to the user with id 1. This user does not actually have to exist, so a stand-alone backend can be used to manually recover from certain kinds of accidental damage to the system catalogs. Implicit superuser powers are granted to the user with id 1 in stand-alone mode.

Options

When postgres is started by a postmaster then it inherits all options set by the latter. Additionally, postgres-specific options can be passed from the postmaster with the `-o` switch.

You can avoid having to type these options by setting up a configuration file. See the *Administrator's Guide* for details. Some (safe) options can also be set from the connecting client in an application-dependent way. For example, if the environment variable `PGOPTIONS` is set, then libpq-based clients will pass that string to the server, which will interpret it as postgres command-line options.

General Purpose

The options `-A`, `-B`, `-c`, `-d`, `-D`, `-F`, and `--name` have the same meanings as for the postmaster.

-e

Sets the default date style to “European”, which means that the “day before month” (rather than month before day) rule is used to interpret ambiguous date input, and that the day is printed before the month in certain date output formats. See the *PostgreSQL User's Guide* for more information.

-o *filename*

Sends all debugging and error output to *filename*. If the backend is running under the postmaster, this option is ignored, and the stderr inherited from the postmaster is used.

-P

Ignore system indexes while scanning/updating system tuples. The REINDEX command for system tables/indexes requires this option to be used.

-s

Print time information and other statistics at the end of each query. This is useful for benchmarking or for use in tuning the number of buffers.

-S *sort-mem*

Specifies the amount of memory to be used by internal sorts and hashes before resorting to temporary disk files. The value is specified in kilobytes, and defaults to 512 kilobytes. Note that for a complex query, several sorts and/or hashes might be running in parallel, and each one will be allowed to use as much as *sort-mem* kilobytes before it starts to put data into temporary files.

Options for stand-alone mode

database

Specifies the name of the database to be accessed. If it is omitted it defaults to the user name.

-E

Echo all queries.

-N

Disables use of newline as a query delimiter.

Semi-internal Options

There are several other options that may be specified, used mainly for debugging purposes. These are listed here only for the use by PostgreSQL system developers. *Use of any of these options is highly discouraged.* Furthermore, any of these options may disappear or change in a future release without notice.

-f { s | i | m | n | h }

Forbids the use of particular scan and join methods: s and i disable sequential and index scans respectively, while n, m, and h disable nested-loop, merge and hash joins respectively.

Note

Neither sequential scans nor nested-loop joins can be disabled completely; the -fs and -fn options simply discourage the optimizer from using those plan types if it has any other alternative.

-i

Prevents query execution, but shows the plan tree.

-O

Allows the structure of system tables to be modified. This is used by initdb.

-p *database*

Indicates that this server has been started by a postmaster and makes different assumptions about buffer pool management, file descriptors, etc.

-t pa[rser] | pl[anner] | e[xecutor]

Print timing statistics for each query relating to each of the major system modules. This option cannot be used together with the `-s` option.

-v *protocol*

Specifies the version number of the frontend/backend protocol to be used for this particular session.

-W *seconds*

As soon as this option is encountered, the process sleeps for the specified amount of seconds. This gives developers time to attach a debugger to the backend process.

Usage

Start a stand-alone backend with a command like

postgres -D $PGDATA *other-options* my_database

Provide the correct path to the database area with `-D`, or make sure that the environment variable `PGDATA` is set. Also specify the name of the particular database you want to work in.

Normally, the stand-alone backend treats newline as the command entry terminator; there is no intelligence about semicolons, as there is in psql. To continue a command across multiple lines, you must type backslash just before each newline except the last one.

But if you use the `-N` command line switch, then newline does not terminate command entry. The backend will read the standard input until the end-of-file (EOF) marker, then process the input as a single query string. Backslash-newline is not treated specially in this case.

To quit the session, type EOF (**Control**+**D**, usually). If you've used `-N`, two consecutive EOFs are needed to exit.

Note that the stand-alone backend does not provide sophisticated line-editing features (no command history, for example).

See Also

initdb, ipcclean, postmaster

<docinfo>2000-12-25</docinfo>

Name

postmaster — PostgreSQL multiuser database server

Synopsis

`postmaster` [-A [0] | [1]] [-B *`nbuffers`*] [-c *`name=value`*] [-d *`debug-level`*] [-D *`datadir`*] [-F] [-h *`hostname`*] [-i] [-k *`directory`*] [-l] [-N *`max-connections`*] [-o *`extra-options`*] [-p *`port`*] [-S] [--*`name=value`*] [[-n] | [-s]]

Description

postmaster is the PostgreSQL multiuser database server. In order for a client application to access a database it connects (over a network or locally) to a running postmaster. The postmaster then starts a separate server process ("postgres") to handle the connection. The postmaster also manages the communication among server processes.

By default the postmaster starts in the foreground and prints log messages to the standard output. In practical applications the postmaster should be started as a background process, perhaps at boot time.

One postmaster always manages the data from exactly one database cluster. A database cluster is a collection of databases that is stored at a common file system location. When the postmaster starts it needs to know the location of the database cluster files ("data area"). This is done with the `-D` invocation option or the `PGDATA` environment variable; there is no default. More than one postmaster process can run on a system at one time, as long as they use different data areas and different communication ports (see below). A data area is created with initdb.

Options

postmaster accepts the following command line arguments. For a detailed discussion of the options consult the *Administrator's Guide*. You can also save typing most of these options by setting up a configuration file.

-A 0|1

Enables run-time assert checks, which is a debugging aid to detect programming mistakes. This is only available if it was enabled during compilation. If so, the default is on.

-B *`nbuffers`*

Sets the number of shared buffers for use by the server processes. This value defaults to 64 buffers, where each buffer is 8 kB.

-c *`name=value`*

Sets a named run-time parameter. Consult the *Administrator's Guide* for a list and descriptions. Most of the other command line options are in fact short forms of such a parameter assignment. `-c` can appear multiple times to set multiple parameters.

-d *`debug-level`*

Sets the debug level. The higher this value is set, the more debugging output is written to the server log. The default is 0, which means no debugging. Values up to 4 are useful; higher numbers produce no additional output.

-D *datadir*

Specifies the file system location of the data directory. See discussion above.

-F

Disables `fsync` calls for performance improvement, at the risk of data corruption in event of a system crash. Read the detailed documentation before using this!

-h *hostname*

Specifies the TCP/IP host name or address on which the postmaster is to listen for connections from client applications. Defaults to listening on all configured addresses (including localhost).

-i

Allows clients to connect via TCP/IP (Internet domain) connections. Without this option, only local Unix domain socket connections are accepted.

-k *directory*

Specifies the directory of the Unix-domain socket on which the postmaster is to listen for connections from client applications. The default is normally `/tmp`, but can be changed at build time.

-l

Enables secure connections using SSL. The `-i` option is also required. You must have compiled with SSL enabled to use this option.

-N *max-connections*

Sets the maximum number of client connections that this postmaster will accept. By default, this value is 32, but it can be set as high as your system will support. (Note that `-B` is required to be at least twice `-N`. See the *Administrator's Guide* for a discussion of system resource requirements for large numbers of client connections.)

-o *extra-options*

The command line-style options specified in *extra-options* are passed to all backend server processes started by this postmaster. See postgres for possibilities. If the option string contains any spaces, the entire string must be quoted.

-p *port*

Specifies the TCP/IP port or local Unix domain socket file extension on which the postmaster is to listen for connections from client applications. Defaults to the value of the `PGPORT` environment variable, or if `PGPORT` is not set, then defaults to the value established during compilation (normally 5432). If you specify a port other than the default port, then all client applications must specify the same port using either command-line options or `PGPORT`.

-S

Specifies that the postmaster process should start up in silent mode. That is, it will disassociate from the user's (controlling) terminal, start its own process group, and redirect its standard output and standard error to `/dev/null`.

Using this switch discards all logging output, which is probably not what you want, since it makes it very difficult to troubleshoot problems. See below for a better way to start the postmaster in the background.

--name=value

Sets a named run-time parameter; a shorter form of `-c`.

Two additional command line options are available for debugging problems that cause a backend to die abnormally. These options control the behavior of the postmaster in this situation, and *neither option is intended for use in ordinary operation*.

The ordinary strategy for this situation is to notify all other backends that they must terminate and then reinitialize the shared memory and semaphores. This is because an errant backend could have corrupted some shared state before terminating.

These special-case options are:

-n

postmaster will not reinitialize shared data structures. A knowledgeable system programmer can then use a debugger to examine shared memory and semaphore state.

-s

postmaster will stop all other backend processes by sending the signal `SIGSTOP`, but will not cause them to terminate. This permits system programmers to collect core dumps from all backend processes by hand.

Outputs

`semget: No space left on device`

If you see this message, you should run the ipcclean command. After doing so, try starting postmaster again. If this still doesn't work, you probably need to configure your kernel for shared memory and semaphores as described in the installation notes. If you run multiple instances of postmaster on a single host, or have a kernel with particularly small shared memory and/or semaphore limits, you may have to reconfigure your kernel to increase its shared memory or semaphore parameters.

Tip

You may be able to postpone reconfiguring your kernel by decreasing `-B` to reduce the shared memory consumption of PostgreSQL, and/or by reducing `-N` to reduce the semaphore consumption.

`StreamServerPort: cannot bind to port`

If you see this message, you should make certain that there is no other postmaster process already running on the same port number. The easiest way to determine this is by using the command

```
$ ps ax | grep postmaster
```

or

```
$ ps -e | grep postmaster
```

depending on your system.

If you are sure that no other postmaster processes are running and you still get this error, try specifying a different port using the `-p` option. You may also get this error if you terminate the postmaster and

immediately restart it using the same port; in this case, you must simply wait a few seconds until the operating system closes the port before trying again. Finally, you may get this error if you specify a port number that your operating system considers to be reserved. For example, many versions of Unix consider port numbers under 1024 to be *trusted* and only permit the Unix superuser to access them.

Notes

If at all possible, *do not* use `SIGKILL` to kill the postmaster. This will prevent postmaster from freeing the system resources (e.g., shared memory and semaphores) that it holds before terminating.

To terminate the postmaster normally, the signals `SIGTERM`, `SIGINT`, or `SIGQUIT` can be used. The first will wait for all clients to terminate before quitting, the second will forcefully disconnect all clients, and the third will quit immediately without proper shutdown, resulting in a recovery run during restart.

The utility command pg_ctl can be used to start and shut down the postmaster safely and comfortably.

The `--` options will not work on FreeBSD or OpenBSD. Use `-c` instead. This is a bug in the affected operating systems; a future release of PostgreSQL will provide a workaround if this is not fixed.

Usage

To start postmaster in the background using default values, type:

```
$ nohup postmaster >logfile 2>&1 </dev/null &
```

To start postmaster with a specific port:

```
$ postmaster -p 1234
```

This command will start up postmaster communicating through the port 1234. In order to connect to this postmaster using psql, you would need to run it as

```
$ psql -p 1234
```

or set the environment variable `PGPORT`:

```
$ export PGPORT=1234
$ psql
```

Named runtime parameters can be set in either of these styles:

```
$ postmaster -c sort_mem=1234
$ postmaster --sort-mem=1234
```

Either form overrides whatever setting might exist for `sort_mem` in `postgresql.conf`. Notice that underscores in parameter names can be written as either underscore or dash on the command line.

Tip

Except for short-term experiments, it's probably better practice to edit the setting in `postgresql.conf` than to rely on a command-line switch to set a parameter.

PostgreSQL 7.2.8 Developer's Guide

The PostgreSQL Global Development Group

PostgreSQL 7.2.8 Developer's Guide

The PostgreSQL Global Development Group
Copyright © 1996-2001 The PostgreSQL Global Development Group

Abstract

This document contains assorted information that can be of use to PostgreSQL developers.

Legal Notice

PostgreSQL is Copyright © 1996-2001 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.

Table of Contents

1. PostgreSQL Source Code 1
    1.1. Formatting 1
2. Overview of PostgreSQL Internals 2
    2.1. The Path of a Query 2
    2.2. How Connections are Established 2
    2.3. The Parser Stage 3
        2.3.1. Parser 3
        2.3.2. Transformation Process 4
    2.4. The PostgreSQL Rule System 5
        2.4.1. The Rewrite System 5
    2.5. Planner/Optimizer 6
        2.5.1. Generating Possible Plans 6
        2.5.2. Data Structure of the Plan 7
    2.6. Executor 7
3. System Catalogs 9
    3.1. Overview 9
    3.2. pg_aggregate 10
    3.3. pg_attrdef 10
    3.4. pg_attribute 11
    3.5. pg_class 13
    3.6. pg_database 15
    3.7. pg_description 16
    3.8. pg_group 17
    3.9. pg_index 17
    3.10. pg_inherits 18
    3.11. pg_language 19
    3.12. pg_largeobject 20
    3.13. pg_listener 20
    3.14. pg_operator 21
    3.15. pg_proc 22
    3.16. pg_relcheck 23
    3.17. pg_rewrite 23
    3.18. pg_shadow 24
    3.19. pg_statistic 25
    3.20. pg_trigger 26
    3.21. pg_type 27
4. Frontend/Backend Protocol 32
    4.1. Overview 32
    4.2. Protocol 32
        4.2.1. Start-up 32
        4.2.2. Query 34
        4.2.3. Function Call 35
        4.2.4. Notification Responses 36
        4.2.5. Cancelling Requests in Progress 36
        4.2.6. Termination 37
        4.2.7. SSL Session Encryption 37
    4.3. Message Data Types 38
    4.4. Message Formats 38
5. gcc Default Optimizations 46
6. BKI Backend Interface 47
    6.1. BKI File Format 47

6.2. BKI Commands 47
6.3. Example 48
7. Page Files 49
8. Genetic Query Optimization 50
8.1. Query Handling as a Complex Optimization Problem 50
8.2. Genetic Algorithms 50
8.3. Genetic Query Optimization (GEQO) in PostgreSQL 51
8.3.1. Future Implementation Tasks for PostgreSQL GEQO 52
8.4. Further Readings 52
9. Native Language Support 53
9.1. For the Translator 53
9.1.1. Requirements 53
9.1.2. Concepts 53
9.1.3. Creating and maintaining message catalogs 54
9.1.4. Editing the PO files 55
9.2. For the Programmer 55
A. The CVS Repository 58
A.1. Getting The Source Via Anonymous CVS 58
A.2. CVS Tree Organization 59
A.3. Getting The Source Via CVSup 60
A.3.1. Preparing A CVSup Client System 61
A.3.2. Running a CVSup Client 61
A.3.3. Installing CVSup 63
A.3.4. Installation from Sources 64
B. Documentation 66
B.1. DocBook 66
B.2. Toolsets 66
B.2.1. Linux RPM Installation 67
B.2.2. FreeBSD Installation 68
B.2.3. Debian Packages 68
B.2.4. Manual Installation from Source 68
B.3. Building The Documentation 70
B.3.1. HTML 71
B.3.2. Manpages 72
B.3.3. Hardcopy Generation 72
B.3.4. Plain Text Files 74
B.4. Documentation Authoring 74
B.4.1. Emacs/PSGML 74
B.4.2. Other Emacs modes 75

List of Figures

8.1. Structured Diagram of a Genetic Algorithm 51

List of Tables

3.1. System Catalogs 9
3.2. pg_aggregate Columns 10
3.3. pg_attrdef Columns 10
3.4. pg_attribute Columns 11
3.5. pg_class Columns 13
3.6. pg_database Columns 15
3.7. pg_description Columns 17
3.8. pg_group Columns 17
3.9. pg_index Columns 17
3.10. pg_inherits Columns 18
3.11. pg_language Columns 19
3.12. pg_largeobject Columns 20
3.13. pg_listener Columns 20
3.14. pg_operator Columns 21
3.15. pg_proc Columns 22
3.16. pg_relcheck Columns 23
3.17. pg_rewrite Columns 23
3.18. pg_shadow Columns 24
3.19. pg_statistic Columns 25
3.20. pg_trigger Columns 26
3.21. pg_type Columns 27
7.1. Page Layout 49
B.1. Indent Formatting for Table of Contents 73

List of Examples

2.1. A Simple Select 4

Chapter 1. PostgreSQL Source Code

1.1. Formatting

Source code formatting uses a 4 column tab spacing, currently with tabs preserved (i.e. tabs are not expanded to spaces).

For Emacs, add the following (or something similar) to your `~/.emacs` initialization file:

```
;; check for files with a path containing "postgres" or "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 ()
  ;; sets up formatting for PostgreSQL C code
  (interactive)
  (c-mode)
  (setq-default tab-width 4)
  (c-set-style "bsd")             ; set c-basic-offset to 4, plus
 other stuff
  (c-set-offset 'case-label '+)   ; tweak case indent to match PG
 custom
  (setq indent-tabs-mode t))      ; make sure we keep tabs when
 indenting
```

For vi, your `~/.vimrc` or equivalent file should contain the following:

```
set tabstop=4
```

or equivalently from within vi, try

```
:set ts=4
```

The text browsing tools more and less can be invoked as

```
more -x4
less -x4
```

Chapter 2. Overview of PostgreSQL Internals

Author

This chapter originally appeared as a part of [sim98], Stefan Simkovics' Master's Thesis prepared at Vienna University of Technology under the direction of O.Univ.Prof.Dr. Georg Gottlob and Univ.Ass. Mag. Katrin Seyr.

This chapter gives an overview of the internal structure of the backend of PostgreSQL. After having read the following sections you should have an idea of how a query is processed. Don't expect a detailed description here (I think such a description dealing with all data structures and functions used within PostgreSQL would exceed 1000 pages!). This chapter is intended to help understanding the general control and data flow within the backend from receiving a query to sending the results.

2.1. The Path of a Query

Here we give a short overview of the stages a query has to pass in order to obtain a result.

1. A connection from an application program to the PostgreSQL server has to be established. The application program transmits a query to the server and receives the results sent back by the server.

2. The *parser stage* checks the query transmitted by the application program (client) for correct syntax and creates a *query tree*.

3. The *rewrite system* takes the query tree created by the parser stage and looks for any *rules* (stored in the *system catalogs*) to apply to the *querytree* and performs the transformations given in the *rule bodies*. One application of the rewrite system is given in the realization of *views*.

   Whenever a query against a view (i.e. a *virtual table*) is made, the rewrite system rewrites the user's query to a query that accesses the *base tables* given in the *view definition* instead.

4. The *planner/optimizer* takes the (rewritten) querytree and creates a *queryplan* that will be the input to the *executor*.

   It does so by first creating all possible *paths* leading to the same result. For example if there is an index on a relation to be scanned, there are two paths for the scan. One possibility is a simple sequential scan and the other possibility is to use the index. Next the cost for the execution of each plan is estimated and the cheapest plan is chosen and handed back.

5. The executor recursively steps through the *plan tree* and retrieves tuples in the way represented by the plan. The executor makes use of the *storage system* while scanning relations, performs *sorts* and *joins*, evaluates *qualifications* and finally hands back the tuples derived.

In the following sections we will cover every of the above listed items in more detail to give a better understanding on PostgreSQL's internal control and data structures.

2.2. How Connections are Established

PostgreSQL is implemented using a simple "process per-user" client/server model. In this model there is one *client process* connected to exactly one *server process*. As we don't know *per se* how many connections

will be made, we have to use a *master process* that spawns a new server process every time a connection is requested. This master process is called `postmaster` and listens at a specified TCP/IP port for incoming connections. Whenever a request for a connection is detected the `postmaster` process spawns a new server process called `postgres`. The server tasks (`postgres` processes) communicate with each other using *semaphores* and *shared memory* to ensure data integrity throughout concurrent data access. Figure \ref{connection} illustrates the interaction of the master process `postmaster` the server process `postgres` and a client application.

The client process can either be the psql frontend (for interactive SQL queries) or any user application implemented using the `libpg` library. Note that applications implemented using ecpg (the PostgreSQL embedded SQL preprocessor for C) also use this library.

Once a connection is established the client process can send a query to the *backend* (server). The query is transmitted using plain text, i.e. there is no parsing done in the *frontend* (client). The server parses the query, creates an *execution plan*, executes the plan and returns the retrieved tuples to the client by transmitting them over the established connection.

2.3. The Parser Stage

The *parser stage* consists of two parts:

- The *parser* defined in `gram.y` and `scan.l` is built using the Unix tools yacc and lex.
- The *transformation process* does modifications and augmentations to the data structures returned by the parser.

2.3.1. Parser

The parser has to check the query string (which arrives as plain ASCII text) for valid syntax. If the syntax is correct a *parse tree* is built up and handed back otherwise an error is returned. For the implementation the well known Unix tools lex and yacc are used.

The *lexer* is defined in the file `scan.l` and is responsible for recognizing *identifiers*, the *SQL keywords* etc. For every keyword or identifier that is found, a *token* is generated and handed to the parser.

The parser is defined in the file `gram.y` and consists of a set of *grammar rules* and *actions* that are executed whenever a rule is fired. The code of the actions (which is actually C-code) is used to build up the parse tree.

The file `scan.l` is transformed to the C-source file `scan.c` using the program lex and `gram.y` is transformed to `gram.c` using yacc. After these transformations have taken place a normal C-compiler can be used to create the parser. Never make any changes to the generated C-files as they will be overwritten the next time lex or yacc is called.

Note

The mentioned transformations and compilations are normally done automatically using the *makefiles* shipped with the PostgreSQL source distribution.

A detailed description of yacc or the grammar rules given in `gram.y` would be beyond the scope of this paper. There are many books and documents dealing with lex and yacc. You should be familiar with yacc before you start to study the grammar given in `gram.y` otherwise you won't understand what happens there.

For a better understanding of the data structures used in PostgreSQL for the processing of a query we use an example to illustrate the changes made to these data structures in every stage. This example contains the following simple query that will be used in various descriptions and figures throughout the following sections. The query assumes that the tables given in *The Supplier Database* have already been defined.

Example 2.1. A Simple Select

```
select s.sname, se.pno
    from supplier s, sells se
    where s.sno > 2 and s.sno = se.sno;
```

Figure \ref{parsetree} shows the *parse tree* built by the grammar rules and actions given in `gram.y` for the query given in Example 2.1, “A Simple Select” (without the *operator tree* for the *where clause* which is shown in figure \ref{where_clause} because there was not enough space to show both data structures in one figure).

The top node of the tree is a `SelectStmt` node. For every entry appearing in the *from clause* of the SQL query a `RangeVar` node is created holding the name of the *alias* and a pointer to a `RelExpr` node holding the name of the *relation*. All `RangeVar` nodes are collected in a list which is attached to the field `fromClause` of the `SelectStmt` node.

For every entry appearing in the *select list* of the SQL query a `ResTarget` node is created holding a pointer to an `Attr` node. The `Attr` node holds the *relation name* of the entry and a pointer to a `Value` node holding the name of the *attribute*. All `ResTarget` nodes are collected to a list which is connected to the field `targetList` of the `SelectStmt` node.

Figure \ref{where_clause} shows the operator tree built for the where clause of the SQL query given in Example 2.1, “A Simple Select” which is attached to the field `qual` of the `SelectStmt` node. The top node of the operator tree is an `A_Expr` node representing an `AND` operation. This node has two successors called `lexpr` and `rexpr` pointing to two *subtrees*. The subtree attached to `lexpr` represents the qualification `s.sno > 2` and the one attached to `rexpr` represents `s.sno = se.sno`. For every attribute an `Attr` node is created holding the name of the relation and a pointer to a `Value` node holding the name of the attribute. For the constant term appearing in the query a `Const` node is created holding the value.

2.3.2. Transformation Process

The *transformation process* takes the tree handed back by the parser as input and steps recursively through it. If a `SelectStmt` node is found, it is transformed to a `Query` node that will be the top most node of the new data structure. Figure \ref{transformed} shows the transformed data structure (the part for the transformed *where clause* is given in figure \ref{transformed_where} because there was not enough space to show all parts in one figure).

Now a check is made, if the *relation names* in the *FROM clause* are known to the system. For every relation name that is present in the *system catalogs* a RTE node is created containing the relation name, the *alias name* and the *relation id*. From now on the relation ids are used to refer to the *relations* given in the query. All RTE nodes are collected in the *range table entry list* that is connected to the field `rtable` of the `Query` node. If a name of a relation that is not known to the system is detected in the query an error will be returned and the query processing will be aborted.

Next it is checked if the *attribute names* used are contained in the relations given in the query. For every attribute} that is found a TLE node is created holding a pointer to a `Resdom` node (which holds the name of the column) and a pointer to a `VAR` node. There are two important numbers in the `VAR` node. The field `varno` gives the position of the relation containing the current attribute} in the range table entry list

created above. The field `varattno` gives the position of the attribute within the relation. If the name of an attribute cannot be found an error will be returned and the query processing will be aborted.

2.4. The PostgreSQL Rule System

PostgreSQL supports a powerful *rule system* for the specification of *views* and ambiguous *view updates*. Originally the PostgreSQL rule system consisted of two implementations:

- The first one worked using *tuple level* processing and was implemented deep in the *executor*. The rule system was called whenever an individual tuple had been accessed. This implementation was removed in 1995 when the last official release of the PostgreSQL project was transformed into Postgres95.
- The second implementation of the rule system is a technique called *query rewriting*. The *rewrite system*} is a module that exists between the *parser stage* and the *planner/optimizer*. This technique is still implemented.

For information on the syntax and creation of rules in the PostgreSQL system refer to *The PostgreSQL User's Guide*.

2.4.1. The Rewrite System

The *query rewrite system* is a module between the parser stage and the planner/optimizer. It processes the tree handed back by the parser stage (which represents a user query) and if there is a rule present that has to be applied to the query it rewrites the tree to an alternate form.

2.4.1.1. Techniques To Implement Views

Now we will sketch the algorithm of the query rewrite system. For better illustration we show how to implement views using rules as an example.

Let the following rule be given:

```
create rule view_rule
as on select
to test_view
do instead
   select s.sname, p.pname
   from supplier s, sells se, part p
   where s.sno = se.sno and
         p.pno = se.pno;
```

The given rule will be *fired* whenever a select against the relation `test_view` is detected. Instead of selecting the tuples from `test_view` the select statement given in the *action part* of the rule is executed.

Let the following user-query against `test_view` be given:

```
select sname
from test_view
where sname <> 'Smith';
```

Here is a list of the steps performed by the query rewrite system whenever a user-query against `test_view` appears. (The following listing is a very informal description of the algorithm just intended for basic understanding. For a detailed description refer to [ston89]).

`test_view` Rewrite

1. Take the query given in the action part of the rule.

2. Adapt the targetlist to meet the number and order of attributes given in the user-query.

3. Add the qualification given in the where clause of the user-query to the qualification of the query given in the action part of the rule.

Given the rule definition above, the user-query will be rewritten to the following form (Note that the rewriting is done on the internal representation of the user-query handed back by the parser stage but the derived new data structure will represent the following query):

```
select s.sname
from supplier s, sells se, part p
where s.sno = se.sno and
      p.pno = se.pno and
      s.sname <> 'Smith';
```

2.5. Planner/Optimizer

The task of the *planner/optimizer* is to create an optimal execution plan. It first combines all possible ways of *scanning* and *joining* the relations that appear in a query. All the created paths lead to the same result and it's the task of the optimizer to estimate the cost of executing each path and find out which one is the cheapest.

2.5.1. Generating Possible Plans

The planner/optimizer decides which plans should be generated based upon the types of indexes defined on the relations appearing in a query. There is always the possibility of performing a sequential scan on a relation, so a plan using only sequential scans is always created. Assume an index is defined on a relation (for example a B-tree index) and a query contains the restriction `relation.attribute OPR constant`. If `relation.attribute` happens to match the key of the B-tree index and `OPR` is anything but '<>' another plan is created using the B-tree index to scan the relation. If there are further indexes present and the restrictions in the query happen to match a key of an index further plans will be considered.

After all feasible plans have been found for scanning single relations, plans for joining relations are created. The planner/optimizer considers only joins between every two relations for which there exists a corresponding join clause (i.e. for which a restriction like `where rel1.attr1=rel2.attr2` exists) in the where qualification. All possible plans are generated for every join pair considered by the planner/optimizer. The three possible join strategies are:

- *nested iteration join*: The right relation is scanned once for every tuple found in the left relation. This strategy is easy to implement but can be very time consuming.

- *merge sort join*: Each relation is sorted on the join attributes before the join starts. Then the two relations are merged together taking into account that both relations are ordered on the join attributes. This kind of join is more attractive because every relation has to be scanned only once.

- *hash join*: the right relation is first hashed on its join attributes. Next the left relation is scanned and the appropriate values of every tuple found are used as hash keys to locate the tuples in the right relation.

2.5.2. Data Structure of the Plan

Here we will give a little description of the nodes appearing in the plan. Figure \ref{plan} shows the plan produced for the query in example \ref{simple_select}.

The top node of the plan is a `MergeJoin` node that has two successors, one attached to the field `lefttree` and the second attached to the field `righttree`. Each of the subnodes represents one relation of the join. As mentioned above a merge sort join requires each relation to be sorted. That's why we find a `Sort` node in each subplan. The additional qualification given in the query (`s.sno > 2`) is pushed down as far as possible and is attached to the `qpqual` field of the leaf `SeqScan` node of the corresponding subplan.

The list attached to the field `mergeclauses` of the `MergeJoin` node contains information about the join attributes. The values `65000` and `65001` for the `varno` fields in the `VAR` nodes appearing in the `mergeclauses` list (and also in the `targetlist`) mean that not the tuples of the current node should be considered but the tuples of the next "deeper" nodes (i.e. the top nodes of the subplans) should be used instead.

Note that every `Sort` and `SeqScan` node appearing in figure \ref{plan} has got a `targetlist` but because there was not enough space only the one for the `MergeJoin` node could be drawn.

Another task performed by the planner/optimizer is fixing the *operator ids* in the `Expr` and `Oper` nodes. As mentioned earlier, PostgreSQL supports a variety of different data types and even user defined types can be used. To be able to maintain the huge amount of functions and operators it is necessary to store them in a system table. Each function and operator gets a unique operator id. According to the types of the attributes used within the qualifications etc., the appropriate operator ids have to be used.

2.6. Executor

The *executor* takes the plan handed back by the planner/optimizer and starts processing the top node. In the case of our example (the query given in example \ref{simple_select}) the top node is a `MergeJoin` node.

Before any merge can be done two tuples have to be fetched (one from each subplan). So the executor recursively calls itself to process the subplans (it starts with the subplan attached to `lefttree`). The new top node (the top node of the left subplan) is a `SeqScan` node and again a tuple has to be fetched before the node itself can be processed. The executor calls itself recursively another time for the subplan attached to `lefttree` of the `SeqScan` node.

Now the new top node is a `Sort` node. As a sort has to be done on the whole relation, the executor starts fetching tuples from the `Sort` node's subplan and sorts them into a temporary relation (in memory or a file) when the `Sort` node is visited for the first time. (Further examinations of the `Sort` node will always return just one tuple from the sorted temporary relation.)

Every time the processing of the `Sort` node needs a new tuple the executor is recursively called for the `SeqScan` node attached as subplan. The relation (internally referenced by the value given in the `scanrelid` field) is scanned for the next tuple. If the tuple satisfies the qualification given by the tree attached to `qpqual` it is handed back, otherwise the next tuple is fetched until the qualification is satisfied. If the last tuple of the relation has been processed a `NULL` pointer is returned.

After a tuple has been handed back by the `lefttree` of the `MergeJoin` the `righttree` is processed in the same way. If both tuples are present the executor processes the `MergeJoin` node. Whenever a new tuple from one of the subplans is needed a recursive call to the executor is performed to obtain it. If a joined tuple could be created it is handed back and one complete processing of the plan tree has finished.

Now the described steps are performed once for every tuple, until a `NULL` pointer is returned for the processing of the `MergeJoin` node, indicating that we are finished.

Chapter 3. System Catalogs

3.1. Overview

The system catalogs are the place where a relational database management system stores schema metadata, such as information about tables and columns, and internal bookkeeping information. PostgreSQL's system catalogs are regular tables. You can drop and recreate the tables, add columns, insert and update values, and severely mess up your system that way. Normally one should not change the system catalogs by hand, there are always SQL commands to do that. (For example, `CREATE DATABASE` inserts a row into the `pg_database` catalog -- and actually creates the database on disk.) There are some exceptions for esoteric operations, such as adding index access methods.

Table 3.1. System Catalogs

Catalog Name	Purpose
pg_aggregate	aggregate functions
pg_am	index access methods
pg_amop	access method operators
pg_amproc	access method support procedures
pg_attrdef	column default values
pg_attribute	table columns (“attributes”, “fields”)
pg_class	tables, indexes, sequences (“relations”)
pg_database	databases within this database cluster
pg_description	descriptions or comments on database objects
pg_group	groups of database users
pg_index	additional index information
pg_inherits	table inheritance hierarchy
pg_language	languages for writing functions
pg_largeobject	large objects
pg_listener	asynchronous notification
pg_opclass	index access method operator classes
pg_operator	operators
pg_proc	functions and procedures
pg_relcheck	check constraints
pg_rewrite	query rewriter rules
pg_shadow	database users
pg_statistic	optimizer statistics
pg_trigger	triggers
pg_type	data types

More detailed documentation of most catalogs follow below. The catalogs that relate to index access methods are explained in the *Programmer's Guide*.

3.2. pg_aggregate

pg_aggregate stores information about aggregate functions. An aggregate function is a function that operates on a set of values (typically one column from each row that matches a query condition) and returns a single value computed from all these values. Typical aggregate functions are sum, count, and max.

Table 3.2. pg_aggregate Columns

Name	Type	References	Description
aggname	name		Name of the aggregate function
aggowner	int4	pg_shadow.usesysid	Owner (creator) of the aggregate function
aggtransfn	regproc (function)	pg_proc.oid	Transition function
aggfinalfn	regproc (function)	pg_proc.oid	Final function
aggbasetype	oid	pg_type.oid	The input datatype for this aggregate function
aggtranstype	oid	pg_type.oid	The type of the aggregate function's internal transition (state) data
aggfinaltype	oid	pg_type.oid	The type of the result
agginitval	text		The initial value of the transition state. This is a text field containing the initial value in its external string representation. If the field is NULL, the transition state value starts out NULL.

New aggregate functions are registered with the CREATE AGGREGATE command. See the *Programmer's Guide* for more information about writing aggregate functions and the meaning of the transition functions, etc.

An aggregate function is identified through name *and* argument type. Hence aggname and aggbasetype are the composite primary key.

3.3. pg_attrdef

This catalog stores column default values. The main information about columns is stored in pg_attribute (see below). Only columns that explicitly specify a default value (when the table is created or the column is added) will have an entry here.

Table 3.3. pg_attrdef Columns

Name	Type	References	Description
adrelid	oid	pg_class.oid	The table this column belongs to

Name	Type	References	Description
adnum	int2	pg_attribute.attnum	The number of the column
adbin	text		An internal representation of the column default value
adsrc	text		A human-readable representation of the default value

3.4. pg_attribute

pg_attribute stores information about table columns. There will be exactly one pg_attribute row for every column in every table in the database. (There will also be attribute entries for indexes and other objects. See pg_class.)

The term attribute is equivalent to column and is used for historical reasons.

Table 3.4. pg_attribute Columns

Name	Type	References	Description
attrelid	oid	pg_class.oid	The table this column belongs to
attname	name		Column name
atttypid	oid	pg_type.oid	The data type of this column
attstattarget	int4		attstattarget controls the level of detail of statistics accumulated for this column by ANALYZE. A zero value indicates that no statistics should be collected. The exact meaning of positive values is datatype-dependent. For scalar datatypes, attstattarget is both the target number of "most common values" to collect, and the target number of histogram bins to create.
attlen	int2		This is a copy of the pg_type.typlen for this column's type.
attnum	int2		The number of the column. Ordinary columns are numbered

Name	Type	References	Description
			from 1 up. System columns, such as `oid`, have (arbitrary) negative numbers.
attndims	`int4`		Number of dimensions, if the column is an array type; otherwise 0. (Presently, the number of dimensions of an array is not enforced, so any nonzero value effectively means “it's an array”.)
attcacheoff	`int4`		Always -1 in storage, but when loaded into a tuple descriptor in memory this may be updated to cache the offset of the attribute within the tuple.
atttypmod	`int4`		`atttypmod` records type-specific data supplied at table creation time (for example, the maximum length of a `varchar` column). It is passed to type-specific input and output functions as the third argument. The value will generally be -1 for types that do not need typmod.
attbyval	`bool`		A copy of `pg_type.typbyval` of this column's type
attstorage	`char`		A copy of `pg_type.typstorage` of this column's type
attisset	`bool`		If true, this attribute is a set. In that case, what is really stored in the attribute is the OID of a tuple in the `pg_proc` catalog. The `pg_proc` tuple contains the query string that defines this set - i.e., the query to run to get the set. So the `atttypid` (see above) refers to the type returned by this query, but the actual length of

Name	Type	References	Description
			this attribute is the length (size) of an `oid`. --- At least this is the theory. All this is probably quite broken these days.
attalign	`char`		A copy of `pg_type.typalign` of this column's type
attnotnull	`bool`		This represents a NOT NULL constraint. It is possible to change this field to enable or disable the constraint.
atthasdef	`bool`		This column has a default value, in which case there will be a corresponding entry in the `pg_attrdef` catalog that actually defines the value.

3.5. pg_class

`pg_class` catalogues tables and mostly everything else that has columns or is otherwise similar to a table. This includes indexes (but see also `pg_index`), sequences, views, and some kinds of special relation. Below, when we mean all of these kinds of objects we speak of “relations”. Not all fields are meaningful for all relation types.

Table 3.5. pg_class Columns

Name	Type	References	Description
relname	`name`		Name of the table, index, view, etc.
reltype	`oid`	pg_type.oid	The OID of the data type that corresponds to this table, if any (zero for indexes, which have no pg_type entry)
relowner	`int4`	pg_shadow.usesysid	Owner of the relation
relam	`oid`	pg_am.oid	If this is an index, the access method used (btree, hash, etc.)
relfilenode	`oid`		Name of the on-disk file of this relation
relpages	`int4`		Size of the on-disk representation of this table in pages (size `BLCKSZ`). This is only an estimate used by the

Name	Type	References	Description
			planner. It is updated by VACUUM, ANALYZE, and CREATE INDEX.
reltuples	float4		Number of tuples in the table. This is only an estimate used by the planner. It is updated by VACUUM, ANALYZE, and CREATE INDEX.
reltoastrelid	oid	pg_class.oid	Oid of the TOAST table associated with this table, 0 if none. The TOAST table stores large attributes “out of line” in a secondary table.
reltoastidxid	oid	pg_class.oid	For a TOAST table, the OID of its index. 0 if not a TOAST table.
relhasindex	bool		True if this is a table and it has (or recently had) any indexes. This is set by CREATE INDEX, but not cleared immediately by DROP INDEX. VACUUM clears relhasindex if it finds the table has no indexes.
relisshared	bool		True if this table is shared across all databases in the cluster. Only certain system catalogs (such as pg_database) are shared.
relkind	char		'r' = ordinary table, 'i' = index, 'S' = sequence, 'v' = view, 's' = special, 't' = secondary TOAST table
relnatts	int2		Number of user columns in the relation (system columns not counted). There must be this many corresponding entries in pg_attribute. See also pg_attribute.attnum.
relchecks	int2		Number of check constraints on the table;

Name	Type	References	Description
			see pg_relcheck catalog
reltriggers	int2		Number of triggers on the table; see pg_trigger catalog
relukeys	int2		unused (*Not* the number of unique keys)
relfkeys	int2		unused (*Not* the number of foreign keys on the table)
relrefs	int2		unused
relhasoids	bool		True if we generate an OID for each row of the relation.
relhaspkey	bool		True if the table has (or once had) a primary key.
relhasrules	bool		Table has rules; see pg_rewrite catalog
relhassubclass	bool		At least one table inherits from this one
relacl	aclitem[]		Access permissions. See the descriptions of GRANT and REVOKE for details.

3.6. pg_database

The pg_database catalog stores information about the available databases. Databases are created with the CREATE DATABASE command. Consult the *Administrator's Guide* for details about the meaning of some of the parameters.

Unlike most system catalogs, pg_database is shared across all databases of a cluster: there is only one copy of pg_database per cluster, not one per database.

Table 3.6. pg_database Columns

Name	Type	References	Description
datname	name		Database name
datdba	int4	pg_shadow.usesysid	Owner of the database, initially who created it
encoding	int4		Character/multibyte encoding for this database
datistemplate	bool		If true then this database can be used in the "TEMPLATE" clause of CREATE DATABASE to

Name	Type	References	Description
			create the new database as a clone of this one.
datallowconn	bool		If false then no one can connect to this database. This is used to protect the template0 database from being altered.
datlastsysoid	oid		Last system OID in the database; useful particularly to pg_dump
datvacuumxid	xid		All tuples inserted or deleted by transaction IDs before this one have been marked as known committed or known aborted in this database. This is used to determine when commit-log space can be recycled.
datfrozenxid	xid		All tuples inserted by transaction IDs before this one have been relabeled with a permanent ("frozen") transaction ID in this database. This is useful to check whether a database must be vacuumed soon to avoid transaction ID wraparound problems.
datpath	text		If the database is stored at an alternative location then this records the location. It's either an environment variable name or an absolute path, depending how it was entered.

3.7. pg_description

The pg_description table can store an optional description or comment for each database object. Descriptions can be manipulated with the COMMENT command. Client applications can view the descriptions by joining with this table. Many builtin system objects have comments associated with them that are shown by psql's \d commands.

Table 3.7. pg_description Columns

Name	Type	References	Description
objoid	oid	any oid attribute	The oid of the object this description pertains to
classoid	oid	pg_class.oid	The oid of the system catalog this object appears in
objsubid	int4		For a comment on a table attribute, this is the attribute's column number (the objoid and classoid refer to the table itself). For all other object types, this field is presently zero.
description	text		Arbitrary text that serves as the description of this object.

3.8. pg_group

This catalog defines groups and stores what users belong to what groups. Groups are created with the CREATE GROUP command. Consult the *Administrator's Guide* for information about user permission management.

Because user and group identities are cluster-wide, pg_group is shared across all databases of a cluster: there is only one copy of pg_group per cluster, not one per database.

Table 3.8. pg_group Columns

Name	Type	References	Description
groname	name		Name of the group
grosysid	int4		An arbitrary number to identify this group
grolist	int4[]	pg_shadow.usesysid	An array containing the ids of the users in this group

3.9. pg_index

pg_index contains part of the information about indexes. The rest is mostly in pg_class.

Table 3.9. pg_index Columns

Name	Type	References	Description
indexrelid	oid	pg_class.oid	The oid of the pg_class entry for this index

Name	Type	References	Description
indrelid	oid	pg_class.oid	The oid of the pg_class entry for the table this index is for
indproc	regproc	pg_proc.oid	The registered procedure if this is a functional index
indkey	int2vector	pg_attribute.attnum	This is a vector (array) of up to INDEX_MAX_KEYS values that indicate which table columns this index pertains to. For example a value of 1 3 would mean that the first and the third column make up the index key.
indclass	oidvector	pg_opclass.oid	For each column in the index key this contains a reference to the "operator class" to use. See pg_opclass for details.
indisclustered	bool		unused
indisunique	bool		If true, this is a unique index.
indisprimary	bool		If true, this index represents the primary key of the table. (indisunique should always be true when this is true.)
indreference	oid		unused
indpred	text		Expression tree (in the form of a nodeToString representation) for partial index predicate

3.10. pg_inherits

This catalog records information about table inheritance hierarchies.

Table 3.10. pg_inherits Columns

Name	Type	References	Description
inhrelid	oid	pg_class.oid	This is the reference to the subtable, that is, it records the fact that the identified table

Name	Type	References	Description
			is inherited from some other table.
inhparent	oid	pg_class.oid	This is the reference to the parent table, which the table referenced by inhrelid inherited from.
inhseqno	int4		If there is more than one parent for a subtable (multiple inheritance), this number tells the order in which the inherited columns are to be arranged. The count starts at 1.

3.11. pg_language

pg_language registers call interfaces or languages in which you can write functions or stored procedures. See under CREATE LANGUAGE and in the *Programmer's Guide* for more information about language handlers.

Table 3.11. pg_language Columns

Name	Type	References	Description
lanname	name		Name of the language (to be specified when creating a function)
lanispl	bool		This is false for internal languages (such as SQL) and true for dynamically loaded language handler modules. It essentially means that, if it is true, the language may be dropped.
lanpltrusted	bool		This is a trusted language. See under CREATE LANGUAGE what this means. If this is an internal language (lanispl is false) then this field is meaningless.
lanplcallfoid	oid	pg_proc.oid	For non-internal languages this references the language handler, which is a special function that is responsible for executing all functions that are

Name	Type	References	Description
			written in the particular language.
lancompiler	text		not currently used

3.12. pg_largeobject

pg_largeobject holds the data making up “large objects”. A large object is identified by an OID assigned when it is created. Each large object is broken into segments or “pages” small enough to be conveniently stored as rows in pg_largeobject. The amount of data per page is defined to be LOBLKSIZE (which is currently BLCKSZ/4, or typically 2Kbytes).

Table 3.12. pg_largeobject Columns

Name	Type	References	Description
loid	oid		Identifier of the large object that includes this page
pageno	int4		Page number of this page within its large object (counting from zero)
data	bytea		Actual data stored in the large object. This will never be more than LOBLKSIZE bytes, and may be less.

Each row of pg_largeobject holds data for one page of a large object, beginning at byte offset (pageno * LOBLKSIZE) within the object. The implementation allows sparse storage: pages may be missing, and may be shorter than LOBLKSIZE bytes even if they are not the last page of the object. Missing regions within a large object read as zeroes.

3.13. pg_listener

pg_listener supports the LISTEN and NOTIFY commands. A listener creates an entry in pg_listener for each notification name it is listening for. A notifier scans pg_listener and updates each matching entry to show that a notification has occurred. The notifier also sends a signal (using the PID recorded in the table) to awaken the listener from sleep.

Table 3.13. pg_listener Columns

Name	Type	References	Description
relname	name		Notify condition name. (The name need not match any actual relation in the database; the term “relname” is historical.)
listenerpid	int4		PID of the backend process that created this entry.

Name	Type	References	Description
notification	int4		Zero if no event is pending for this listener. If an event is pending, the PID of the backend that sent the notification.

3.14. pg_operator

See CREATE OPERATOR and the *Programmer's Guide* for details on these operator parameters.

Table 3.14. pg_operator Columns

Name	Type	References	Description
oprname	name		Name of the operator
oprowner	int4	pg_shadow.usesysid	Owner (creator) of the operator
oprprec	int2		unused
oprkind	char		'b' = infix (“both”), 'l' = prefix (“left”), 'r' = postfix (“right”)
oprisleft	bool		unused
oprcanhash	bool		This operator supports hash joins.
oprleft	oid	pg_type.oid	Type of the left operand
oprright	oid	pg_type.oid	Type of the right operand
oprresult	oid	pg_type.oid	Type of the result
oprcom	oid	pg_operator.oid	Commutator of this operator, if any
oprnegate	oid	pg_operator.oid	Negator of this operator, if any
oprlsortop	oid	pg_operator.oid	If this operator supports merge joins, the operator that sorts the type of the left-hand operand
oprrsortop	oid	pg_operator.oid	If this operator supports merge joins, the operator that sorts the type of the right-hand operand
oprcode	regproc		Function that implements this operator
oprrest	regproc		Restriction selectivity estimation function for this operator
oprjoin	regproc		Join selectivity estimation function for this operator

3.15. pg_proc

This catalog stores information about functions (or procedures). The description of `CREATE FUNCTION` and the *Programmer's Guide* contain more information about the meaning of some fields.

Table 3.15. pg_proc Columns

Name	Type	References	Description
proname	`name`		Name of the function
proowner	`int4`	pg_shadow.usesysid	Owner (creator) of the function
prolang	`oid`	pg_language.oid	Implementation language or call interface of this function
proisinh	`bool`		unused
proistrusted	`bool`		not functional
proiscachable	`bool`		Function returns same result for same input values
proisstrict	`bool`		Function returns null if any call argument is null. In that case the function won't actually be called at all. Functions that are not “strict” must be prepared to handle null inputs.
pronargs	`int2`		Number of arguments
proretset	`bool`		Function returns a set (ie, multiple values of the specified datatype)
prorettype	`oid`	pg_type.oid	Data type of the return value (0 if the function does not return a value)
proargtypes	`oidvector`	pg_type.oid	A vector with the data types of the function arguments
probyte_pct	`int4`		dead code
properbyte_cpu	`int4`		dead code
propercall_cpu	`int4`		dead code
prooutin_ratio	`int4`		dead code
prosrc	`text`		This tells the function handler how to invoke the function. It might be the actual source code of the function for interpreted languages, a

Name	Type	References	Description
			link symbol, a file name, or just about anything else, depending on the implementation language/call convention.
probin	bytea		Additional information about how to invoke the function. Again, the interpretation is language-specific.

Currently, prosrc contains the function's C-language name (link symbol) for compiled functions, both built-in and dynamically loaded. For all other language types, prosrc contains the function's source text.

Currently, probin is unused except for dynamically-loaded C functions, for which it gives the name of the shared library file containing the function.

3.16. pg_relcheck

This system catalog stores CHECK constraints on tables. (Column constraints are not treated specially. Every column constraint is equivalent to some table constraint.) See under `CREATE TABLE` for more information.

Table 3.16. pg_relcheck Columns

Name	Type	References	Description
rcrelid	oid	pg_class.oid	The table this check constraint is on
rcname	name		Constraint name
rcbin	text		An internal representation of the constraint expression
rcsrc	text		A human-readable representation of the constraint expression

Note

`pg_class.relchecks` needs to match up with the entries in this table.

3.17. pg_rewrite

This system catalog stores rewrite rules for tables and views.

Table 3.17. pg_rewrite Columns

Name	Type	References	Description
rulename	name		Rule name

Name	Type	References	Description
ev_type	char		Event type that the rule is for: '1' = SELECT, '2' = UPDATE, '3' = INSERT, '4' = DELETE
ev_class	oid	pg_class.oid	The table this rule is for
ev_attr	int2		The column this rule is for (currently, always zero to indicate the whole table)
is_instead	bool		True if the rule is an INSTEAD rule
ev_qual	text		Expression tree (in the form of a nodeToString representation) for the rule's qualifying condition
ev_action	text		Query tree (in the form of a nodeToString representation) for the rule's action

Note

pg_class.relhasrules must be true if a table has any rules in this catalog.

3.18. pg_shadow

pg_shadow contains information about database users. The name stems from the fact that this table should not be readable by the public since it contains passwords. pg_user is a publicly readable view on pg_shadow that blanks out the password field.

The *Administrator's Guide* contains detailed information about user and permission management.

Because user identities are cluster-wide, pg_shadow is shared across all databases of a cluster: there is only one copy of pg_shadow per cluster, not one per database.

Table 3.18. pg_shadow Columns

Name	Type	References	Description
usename	name		User name
usesysid	int4		User id (arbitrary number used to reference this user)
usecreatedb	bool		User may create databases
usetrace	bool		not used
usesuper	bool		User is a superuser
usecatupd	bool		User may update system catalogs. (Even a

Name	Type	References	Description
			superuser may not do this unless this attribute is true.)
passwd	text		Password
valuntil	abstime		Account expiry time (only used for password authentication)

3.19. pg_statistic

pg_statistic stores statistical data about the contents of the database. Entries are created by ANALYZE and subsequently used by the query planner. There is one entry for each table column that has been analyzed. Note that all the statistical data is inherently approximate, even assuming that it is up-to-date.

Since different kinds of statistics may be appropriate for different kinds of data, pg_statistic is designed not to assume very much about what sort of statistics it stores. Only extremely general statistics (such as NULL-ness) are given dedicated columns in pg_statistic. Everything else is stored in "slots", which are groups of associated columns whose content is identified by a code number in one of the slot's columns. For more information see src/include/catalog/pg_statistic.h.

pg_statistic should not be readable by the public, since even statistical information about a table's contents may be considered sensitive. (Example: minimum and maximum values of a salary column might be quite interesting.) pg_stats is a publicly readable view on pg_statistic that only exposes information about those tables that are readable by the current user. pg_stats is also designed to present the information in a more readable format than the underlying pg_statistic table --- at the cost that its schema must be extended whenever new slot types are added.

Table 3.19. pg_statistic Columns

Name	Type	References	Description
starelid	oid	pg_class.oid	The table that the described column belongs to
staattnum	int2	pg_attribute.attnum	The number of the described column
stanullfrac	float4		The fraction of the column's entries that are NULL
stawidth	int4		The average stored width, in bytes, of non-NULL entries
stadistinct	float4		The number of distinct non-NULL data values in the column. A value greater than zero is the actual number of distinct values. A value less than zero is the negative of a fraction of the number of rows in

Name	Type	References	Description
			the table (for example, a column in which values appear about twice on the average could be represented by stadistinct = -0.5). A zero value means the number of distinct values is unknown.
stakindN	int2		A code number indicating the kind of statistics stored in the Nth "slot" of the pg_statistic row.
staopN	oid	pg_operator.oid	An operator used to derive the statistics stored in the Nth "slot". For example, a histogram slot would show the < operator that defines the sort order of the data.
stanumbersN	float4[]		Numerical statistics of the appropriate kind for the Nth "slot", or NULL if the slot kind does not involve numerical values.
stavaluesN	text[]		Column data values of the appropriate kind for the Nth "slot", or NULL if the slot kind does not store any data values. For datatype independence, all column data values are converted to external textual form and stored as TEXT datums.

3.20. pg_trigger

This system catalog stores triggers on tables. See under CREATE TRIGGER for more information.

Table 3.20. pg_trigger Columns

Name	Type	References	Description
tgrelid	oid	pg_class.oid	The table this trigger is on
tgname	name		Trigger name (need not be unique)

Name	Type	References	Description
tgfoid	oid	pg_proc.oid	The function to be called
tgtype	int2		Bitmask identifying trigger conditions
tgenabled	bool		True if trigger is enabled (not presently checked everywhere it should be, so disabling a trigger by setting this false does not work reliably)
tgisconstraint	bool		True if trigger is a RI constraint
tgconstrname	name		RI constraint name
tgconstrrelid	oid	pg_class.oid	The table referenced by an RI constraint
tgdeferrable	bool		True if deferrable
tginitdeferred	bool		True if initially deferred
tgnargs	int2		Number of argument strings passed to trigger function
tgattr	int2vector		Currently unused
tgargs	bytea		Argument strings to pass to trigger, each null-terminated

Note

pg_class.reltriggers needs to match up with the entries in this table.

3.21. pg_type

This catalog stores information about datatypes. Scalar types ("base types") are created with CREATE TYPE. A complex type is also created for each table in the database, to represent the row structure of the table.

Table 3.21. pg_type Columns

Name	Type	References	Description
typname	name		Data type name
typowner	int4	pg_shadow.usesysid	Owner (creator) of the type
typlen	int2		Length of the storage representation of the type, -1 if variable length
typprtlen	int2		unused
typbyval	bool		typbyval determines whether internal routines

Name	Type	References	Description
			pass a value of this type by value or by reference. Only `char`, `short`, and `int` equivalent items can be passed by value, so if the type is not 1, 2, or 4 bytes long, PostgreSQL does not have the option of passing by value and so `typbyval` had better be false. Variable-length types are always passed by reference. Note that `typbyval` can be false even if the length would allow pass-by-value; this is currently true for type `float4`, for example.
typtype	`char`		`typtype` is `b` for a base type and `c` for a complex type (i.e., a table's row type). If `typtype` is `c`, `typrelid` is the OID of the type's entry in `pg_class`.
typisdefined	`bool`		True if the type is defined, false if this is a placeholder entry for a not-yet-defined type. When typisdefined is false, nothing except the type name and OID can be relied on.
typdelim	`char`		Character that separates two values of this type when parsing array input. Note that the delimiter is associated with the array element datatype, not the array datatype.
typrelid	`oid`	pg_class.oid	If this is a complex type (see `typtype`), then this field points to the `pg_class` entry that defines the corresponding table. A table could theoretically be used as a composite data type, but this is not fully functional.

Name	Type	References	Description
typelem	oid	pg_type.oid	If typelem is not 0 then it identifies another row in pg_type. The current type can then be subscripted like an array yielding values of type typelem. A "true" array type is variable length (typlen = -1), but some fixed-length (typlen > 0) types also have nonzero typelem, for example name and oidvector. If a fixed-length type has a typelem then its internal representation must be N values of the typelem datatype with no other data. Variable-length array types have a header defined by the array subroutines.
typinput	regproc		Input function
typoutput	regproc		Output function
typreceive	regproc		unused
typsend	regproc		unused
typalign	char		typalign is the alignment required when storing a value of this type. It applies to storage on disk as well as most representations of the value inside PostgreSQL. When multiple values are stored consecutively, such as in the representation of a complete row on disk, padding is inserted before a datum of this type so that it begins on the specified boundary. The alignment reference is the beginning of the first datum in the sequence. Possible values are:

Name	Type	References	Description
			• 'c' = CHAR alignment, i.e., no alignment needed. • 's' = SHORT alignment (2 bytes on most machines). • 'i' = INT alignment (4 bytes on most machines). • 'd' = DOUBLE alignment (8 bytes on many machines, but by no means all). **Note** For types used in system tables, it is critical that the size and alignment defined in `pg_type` agree with the way that the compiler will lay out the field in a struct representing a table row.
typstorage	`char`		`typstorage` tells for variable-length types (those with `typlen` = -1) if the type is prepared for toasting and what the default strategy for attributes of this type should be. Possible values are • 'p': Value must always be stored plain. • 'e': Value can be stored in a "secondary" relation (if relation has one, see `pg_class.reltoastrelid`).

Name	Type	References	Description
			• 'm': Value can be stored compressed inline. • 'x': Value can be stored compressed inline or in “secondary”. Note that 'm' fields can also be moved out to secondary storage, but only as a last resort ('e' and 'x' fields are moved first).
typdefault	`text`		`typdefault` is NULL for types without a default value. If it's not NULL, it contains the external string representation of the type's default value.

Chapter 4. Frontend/Backend Protocol

Note

Written by Phil Thompson (<phil@river-bank.demon.co.uk>). Updates for protocol 2.0 by Tom Lane (<tgl@sss.pgh.pa.us>).

PostgreSQL uses a message-based protocol for communication between frontends and backends. The protocol is implemented over TCP/IP and also on Unix domain sockets. PostgreSQL 6.3 introduced version numbers into the protocol. This was done in such a way as to still allow connections from earlier versions of frontends, but this document does not cover the protocol used by those earlier versions.

This document describes version 2.0 of the protocol, implemented in PostgreSQL 6.4 and later.

Higher level features built on this protocol (for example, how libpq passes certain environment variables after the connection is established) are covered elsewhere.

4.1. Overview

A frontend opens a connection to the server and sends a start-up packet. This includes the names of the user and of the database the user wants to connect to. The server then uses this, and the information in the `pg_hba.conf` file to determine what further authentication information it requires the frontend to send (if any) and responds to the frontend accordingly.

The frontend then sends any required authentication information. Once the server validates this it responds to the frontend that it is authenticated and sends a message indicating successful start-up (normal case) or failure (for example, an invalid database name).

In order to serve multiple clients efficiently, the server launches a new “backend” process for each client. This is transparent to the protocol, however. In the current implementation, a new child process is created immediately after an incoming connection is detected.

When the frontend wishes to disconnect it sends an appropriate packet and closes the connection without waiting for a response from the backend.

Packets are sent as a data stream. The first byte determines what should be expected in the rest of the packet. The exceptions are packets sent as part of the startup and authentication exchange, which comprise a packet length followed by the packet itself. The difference is historical.

4.2. Protocol

This section describes the message flow. There are four different types of flows depending on the state of the connection: start-up, query, function call, and termination. There are also special provisions for notification responses and command cancellation, which can occur at any time after the start-up phase.

4.2.1. Start-up

Initially, the frontend sends a StartupPacket. The server uses this info and the contents of the `pg_hba.conf` file to determine what authentication method the frontend must use. The server then responds with one of the following messages:

ErrorResponse

The server then immediately closes the connection.

AuthenticationOk

The authentication exchange is completed.

AuthenticationKerberosV4

The frontend must then 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 then 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 then send a PasswordPacket 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 then send a PasswordPacket containing the password encrypted via crypt(3), using the 2-character salt specified in the AuthenticationCryptPassword packet. If this is the correct password, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationMD5Password

The frontend must then send a PasswordPacket containing the password encrypted via MD5, using the 4-character salt specified in the AuthenticationMD5Password packet. If this is the correct password, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationSCMCredential

This method 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 should wait for further messages from the server. 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.

ReadyForQuery

Start-up is completed. The frontend may now issue query or function call messages.

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 query cycle. Depending on the coding needs of the frontend, it is reasonable to consider ReadyForQuery as starting a query cycle (and then BackendKeyData indicates successful conclusion of the start-up phase), or to consider ReadyForQuery as ending the start-up phase and each subsequent query cycle.

4.2.2. Query

A Query cycle is initiated by the frontend sending a Query message to the backend. 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 query or function call.

The possible response messages from the backend are:

CompletedResponse

An SQL command completed normally.

CopyInResponse

The backend is ready to copy data from the frontend to a table. The frontend should then send a CopyDataRows message. The backend will then respond with a CompletedResponse message with a tag of `COPY`.

CopyOutResponse

The backend is ready to copy data from a table to the frontend. It then sends a CopyDataRows message, and then a CompletedResponse message with a tag of `COPY`.

CursorResponse

Beginning of the response to a `SELECT`, `FETCH`, `INSERT`, `UPDATE`, or `DELETE` query. In the `FETCH` case the name of the cursor being fetched from is included in the message. Otherwise the message always mentions the “blank” cursor.

RowDescription

Indicates that rows are about to be returned in response to a `SELECT` or `FETCH` query. The message contents describe the layout of the rows. This will be followed by an AsciiRow or BinaryRow message (depending on whether a binary cursor was specified) for each row being returned to the frontend.

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. (CompletedResponse 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` or `FETCH` query normally consists of CursorResponse, RowDescription, zero or more AsciiRow or BinaryRow messages, and finally CompletedResponse. `INSERT`, `UPDATE`, and `DELETE` queries produce CursorResponse followed by CompletedResponse. `COPY` to or from the frontend invokes special protocol as mentioned above. All other query types normally produce only a CompletedResponse 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. (The need to specially distinguish this case is historical.)

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.

A frontend must be prepared to accept ErrorResponse and NoticeResponse messages whenever it is expecting any other type of message.

Actually, it is possible for NoticeResponse to arrive even when the frontend is not expecting any kind of message, that is, the backend is nominally idle. (In particular, the backend can be commanded to terminate by its parent process. In that case it will send a NoticeResponse before closing the connection.) It is recommended that the frontend check for such asynchronous notices just before issuing any new command.

Also, if the frontend issues any `LISTEN` commands then it must be prepared to accept NotificationResponse messages at any time; see below.

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.

4.2.3. Function Call

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.

FunctionResultResponse

The function call was executed and returned a result.

FunctionVoidResponse

The function call was executed and returned no result.

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.

A frontend must be prepared to accept ErrorResponse and NoticeResponse messages whenever it is expecting any other type of message. Also, if it issues any `LISTEN` commands then it must be prepared to accept NotificationResponse messages at any time; see below.

4.2.4. Notification Responses

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.

Notification responses are permitted at any point in the protocol (after start-up), except within another backend message. Thus, the frontend must be prepared to recognize a NotificationResponse message whenever it is expecting any message. Indeed, it should be able to handle NotificationResponse messages even when it is not engaged in a query.

NotificationResponse

A `NOTIFY` command has been executed for a name for which a previous `LISTEN` command was executed. Notifications may be sent at any time.

It may be worth pointing out that the names used in listen and notify commands need not have anything to do with names of relations (tables) in the SQL database. Notification names are simply arbitrarily chosen condition names.

4.2.5. 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 StartupPacket 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.

4.2.6. Termination

The normal, graceful termination procedure is that the frontend sends a Terminate message and immediately closes the connection. On receipt of the message, the backend immediately closes the connection and terminates.

An ungraceful termination may occur due to software failure (i.e., core dump) at either end. 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.

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 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.

4.2.7. SSL Session Encryption

Recent releases of PostgreSQL allow frontend/backend communication to be encrypted using SSL. This provides communication security in environments where attackers might be able to capture the session traffic.

To initiate an SSL-encrypted connection, the frontend initially sends an SSLRequest message rather than a StartupPacket. The server then responds with a single byte containing `Y` 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 `Y`, 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 StartupPacket. In this case the StartupPacket and all subsequent data will be SSL-encrypted. To continue after `N`, send the usual StartupPacket 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.

4.3. Message Data Types

This section describes the base data types used in messages.

Int*n*(*i*)

An *n* bit integer in network byte order. If *i* is specified it is the literal value. Eg. Int16, Int32(42).

LimString*n*(*s*)

A character array of exactly *n* bytes interpreted as a null-terminated string. The zero-byte is omitted if there is insufficient room. If *s* is specified it is the literal value. Eg. LimString32, LimString64("user").

String(*s*)

A conventional C null-terminated string with no length limitation. If *s* is specified it is the literal value. Eg. String, String("user").

Note

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.

Byte*n*(*c*)

Exactly *n* bytes. If *c* is specified it is the literal value. Eg. Byte, Byte1('\n').

4.4. Message Formats

This section describes the detailed format of each message. Each can be sent by either a frontend (F), a backend (B), or both (F & B).

AsciiRow (B)

Byte1('D')

Identifies the message as an ASCII data row. (A prior RowDescription message defines the number of fields in the row and their data types.)

Byte*n*

A bit map with one bit for each field in the row. The 1st field corresponds to bit 7 (MSB) of the 1st byte, the 2nd field corresponds to bit 6 of the 1st byte, the 8th field corresponds to bit 0 (LSB) of the 1st byte, the 9th field corresponds to bit 7 of the 2nd byte, and so on. Each bit is set if the value of the corresponding field is not NULL. If the number of fields is not a multiple of 8, the remainder of the last byte in the bit map is wasted.

Then, for each field with a non-NULL value, there is the following:

Int32

Specifies the size of the value of the field, including this size.

Byte*n*

Specifies the value of the field itself in ASCII characters. *n* is the above size minus 4. There is no trailing zero-byte in the field data; the front end must add one if it wants one.

AuthenticationOk (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(0)

Specifies that the authentication was successful.

AuthenticationKerberosV4 (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(1)

Specifies that Kerberos V4 authentication is required.

AuthenticationKerberosV5 (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(2)

Specifies that Kerberos V5 authentication is required.

AuthenticationCleartextPassword (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(3)

Specifies that a cleartext password is required.

AuthenticationCryptPassword (B)

Byte1('R')

Identifies the message as an authentication request.

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(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(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

The process ID of this backend.

Int32

The secret key of this backend.

BinaryRow (B)

Byte1('B')

Identifies the message as a binary data row. (A prior RowDescription message defines the number of fields in the row and their data types.)

Byte*n*

A bit map with one bit for each field in the row. The 1st field corresponds to bit 7 (MSB) of the 1st byte, the 2nd field corresponds to bit 6 of the 1st byte, the 8th field corresponds to bit 0 (LSB) of the 1st byte, the 9th field corresponds to bit 7 of the 2nd byte, and so on. Each bit is set if the value of the corresponding field is not NULL. If the number of fields is not a multiple of 8, the remainder of the last byte in the bit map is wasted.

Then, for each field with a non-NULL value, there is the following:

Int32

Specifies the size of the value of the field, excluding this size.

Byte*n*

Specifies the value of the field itself in binary format. *n* is the above size.

CancelRequest (F)

Int32(16)

The size of the packet in bytes.

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.

CompletedResponse (B)

Byte1('C')

Identifies the message as a completed response.

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, and *`oid`* is the object ID of the inserted row if *`rows`* is 1, 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.

CopyDataRows (B & F)

This is a stream of rows where each row is terminated by a Byte1('\n'). This is then followed by the sequence Byte1('\\'), Byte1('.'), Byte1('\n').

CopyInResponse (B)

Byte1('G')

Identifies the message as a Start Copy In response. The frontend must now send a CopyDataRows message.

CopyOutResponse (B)

Byte1('H')

Identifies the message as a Start Copy Out response. This message will be followed by a CopyDataRows message.

CursorResponse (B)

Byte1('P')

Identifies the message as a cursor response.

String

The name of the cursor. This will be “blank” if the cursor is implicit.

EmptyQueryResponse (B)

Byte1('I')

Identifies the message as a response to an empty query string.

String("")

Unused.

ErrorResponse (B)

Byte1('E')

Identifies the message as an error.

String

The error message itself.

FunctionCall (F)

Byte1('F')

Identifies the message as a function call.

String("")

Unused.

Int32

Specifies the object ID of the function to call.

Int32

Specifies the number of arguments being supplied to the function.

Then, for each argument, there is the following:

Int32

Specifies the size of the value of the argument, excluding this size.

Byte*n*

Specifies the value of the field itself in binary format. *n* is the above size.

FunctionResultResponse (B)

Byte1('V')

Identifies the message as a function call result.

Byte1('G')

Specifies that a nonempty result was returned.

Int32

Specifies the size of the value of the result, excluding this size.

Byte*n*

Specifies the value of the result itself in binary format. *n* is the above size.

Byte1('0')

Unused. (Strictly speaking, FunctionResultResponse and FunctionVoidResponse are the same thing but with some optional parts to the message.)

FunctionVoidResponse (B)

Byte1('V')

Identifies the message as a function call result.

Byte1('0')

Specifies that an empty result was returned.

NoticeResponse (B)

Byte1('N')

Identifies the message as a notice.

String

The notice message itself.

NotificationResponse (B)

Byte1('A')

Identifies the message as a notification response.

Int32

The process ID of the notifying backend process.

String

The name of the condition that the notify has been raised on.

PasswordPacket (F)

Int32

The size of the packet in bytes.

String

The password (encrypted, if requested).

Query (F)

Byte1('Q')

Identifies the message as a query.

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.

RowDescription (B)

Byte1('T')

Identifies the message as a row description.

Int16

Specifies the number of fields in a row (may be zero).

Then, for each field, there is the following:

String

Specifies the field name.

Int32

Specifies the object ID of the field type.

Int16

Specifies the type size.

Int32

Specifies the type modifier.

SSLRequest (F)

Int32(8)

The size of the packet in bytes.

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.)

StartupPacket (F)

Int32(296)

The size of the packet in bytes.

Int32

The protocol version number. The most significant 16 bits are the major version number. The least 16 significant bits are the minor version number.

LimString64

The database name, defaults to the user name if empty.

LimString32

The user name.

LimString64

Any additional command line arguments to be passed to the backend child process by the server.

LimString64

Unused.

LimString64

The optional tty the backend should use for debugging messages. (Currently, this field is unsupported and ignored.)

Terminate (F)

Byte1('X')

Identifies the message as a termination.

Chapter 5. gcc Default Optimizations

Brian Gallew

Note

Contributed by Brian Gallew (<geek+@cmu.edu>)

Configuring gcc to use certain flags by default is a simple matter of editing the /usr/local/lib/gcc-lib/*platform*/*version*/specs file. The format of this file pretty simple. The file is broken into sections, each of which is three lines long. The first line is "**section_name*:" (e.g. "*asm:"). The second line is a list of flags, and the third line is blank.

The easiest change to make is to append the desired default flags to the list in the appropriate section. As an example, let's suppose that I have linux running on a '486 with gcc 2.7.2 installed in the default location. In the file /usr/local/lib/gcc-lib/i486-linux/2.7.2/specs, 13 lines down I find the following section:

```
- ----------SECTION----------
*cc1:


- ----------SECTION----------
```

As you can see, there aren't any default flags. If I always wanted compiles of C code to use "-m486 -fomit-frame-pointer", I would change it to look like:

```
- ----------SECTION----------
*cc1:
- -m486 -fomit-frame-pointer

- ----------SECTION----------
```

If I wanted to be able to generate 386 code for another, older linux box lying around, I'd have to make it look like this:

```
- ----------SECTION----------
*cc1:
%{!m386:-m486} -fomit-frame-pointer

- ----------SECTION----------
```

This will always omit frame pointers, any will build 486-optimized code unless -m386 is specified on the command line.

You can actually do quite a lot of customization with the specs file. Always remember, however, that these changes are global, and affect all users of the system.

Chapter 6. BKI Backend Interface

Backend Interface (BKI) files are scripts in a special language that are input to the PostgreSQL backend running in the special "bootstrap" mode that allows it to perform database functions without a database system already existing. 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` from some specially formatted C header files in 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.

6.1. 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. You should also study the source code of initdb to get an idea of how the backend is invoked.

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.

6.2. BKI Commands

open *`tablename`*

Open the table called *`tablename`* for further manipulation.

close [*`tablename`*]

Close the open table called *`tablename`*. It is an error if *`tablename`* is not already opened. If no *`tablename`* is given, then the currently open table is closed.

create *`tablename`* (*`name1`* = *`type1`* [, *`name2`* = *`type2`*, ...])

Create a table named *`tablename`* with the columns given in parentheses.

The *`type`* is not necessarily the data type that the column will have in the SQL environment; that is determined by the `pg_attribute` system catalog. The type here is essentially only used to allocate storage. The following types are allowed: `bool`, `bytea`, `char` (1 byte), `name`, `int2`, `int2vector`, `int4`, `regproc`, `text`, `oid`, `tid`, `xid`, `cid`, `oidvector`, `smgr`, `_int4` (array), `_aclitem` (array). Array types can also be indicated by writing `[]` after the name of the element type.

Note

The table will only be created on disk, it will not automatically be registered in the system catalogs and will therefore not be accessible unless appropriate rows are inserted in `pg_class`, `pg_attribute`, etc.

insert [OID = *`oid_value`*] (*`value1 value2 ...`*)

Insert a new row into the open table using *`value1`*, *`value2`*, etc., for its column values and *`oid_value`* for its OID. If *`oid_value`* 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 *`tablename`* 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.

build indices

Build the indices that have previously been declared.

6.3. Example

The following sequence of commands will create the `test_table` table with the 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
```

Chapter 7. Page Files

A description of the database file default page format.

This section provides an overview of the page format used by PostgreSQL tables. User-defined access methods need not use this page format.

In the following explanation, a *byte* is assumed to contain 8 bits. In addition, the term *item* refers to data that is stored in PostgreSQL tables.

Table 7.1, “Page Layout” shows how pages in both normal PostgreSQL tables and PostgreSQL indexes (e.g., a B-tree index) are structured.

Table 7.1. Sample Page Layout

Item	Description
itemPointerData	
filler	
itemData...	
Unallocated Space	
ItemContinuationData	
Special Space	
“ItemData 2”	
“ItemData 1”	
ItemIdData	
PageHeaderData	

The first 8 bytes of each page consists of a page header (PageHeaderData). Within the header, the first three 2-byte integer fields (*lower*, *upper*, and *special*) represent byte offsets to the start of unallocated space, to the end of unallocated space, and to the start of *special space*. Special space is a region at the end of the page that is allocated at page initialization time and contains information specific to an access method. The last 2 bytes of the page header, *opaque*, encode the page size and information on the internal fragmentation of the page. Page size is stored in each page because frames in the buffer pool may be subdivided into equal sized pages on a frame by frame basis within a table. The internal fragmentation information is used to aid in determining when page reorganization should occur.

Following the page header are item identifiers (*ItemIdData*). New item identifiers are allocated from the first four bytes of unallocated space. Because an item identifier is never moved until it is freed, its index may be used to indicate the location of an item on a page. In fact, every pointer to an item (*ItemPointer*) created by PostgreSQL consists of a frame number and an index of an item identifier. An item identifier contains a byte-offset to the start of an item, its length in bytes, and a set of attribute bits which affect its interpretation.

The items themselves are stored in space allocated backwards from the end of unallocated space. Usually, the items are not interpreted. However when the item is too long to be placed on a single page or when fragmentation of the item is desired, the item is divided and each piece is handled as distinct items in the following manner. The first through the next to last piece are placed in an item continuation structure (*ItemContinuationData*). This structure contains itemPointerData which points to the next piece and the piece itself. The last piece is handled normally.

Chapter 8. Genetic Query Optimization

Martin Utesch, University of Mining and Technology

Author

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.

8.1. 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 query optimization technique is inadequate to support database application domains that involve the need for extensive queries, such as artificial intelligence.

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 being developed.

In the following we propose the implementation of a *Genetic Algorithm* as an option for the database query optimization problem.

8.2. Genetic Algorithms

The genetic algorithm (GA) is a heuristic optimization method which operates through determined, 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*.

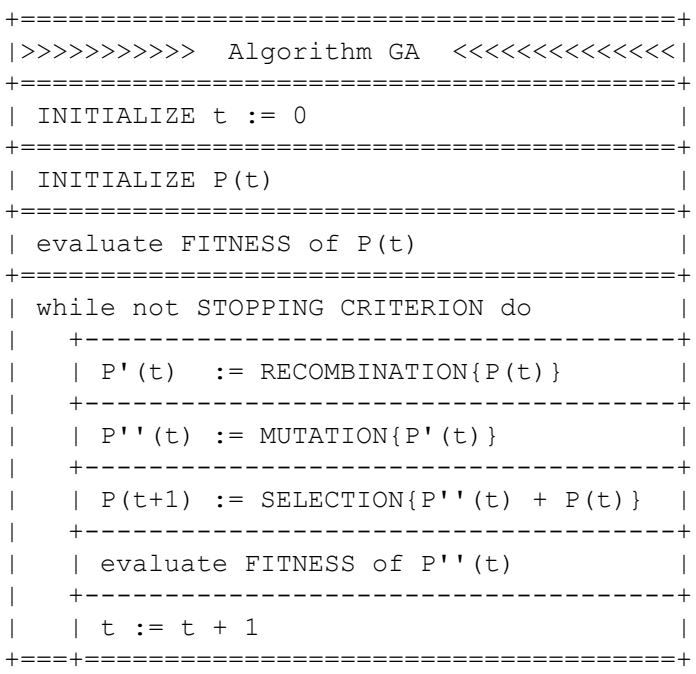
The coordinates of an individual in the search space are represented by *chromosomes*, in essence a set of character strings. A *gene* is a subsection of a chromosome which encodes the value of a single parameter being optimized. Typical encodings for a gene could be *binary* or *integer*.

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).

Figure 8.1. Structured Diagram of a Genetic Algorithm

P(t)	generation of ancestors at a time t
P"(t)	generation of descendants at a time t



8.3. Genetic Query Optimization (GEQO) in PostgreSQL

The GEQO module is intended for the solution of the query optimization problem similar to a 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. E. g., the query tree

```
   /\
  /\ 2
 /\ 3
4  1
```

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.

8.3.1. Future Implementation Tasks for PostgreSQL GEQO

Work is still needed to improve the genetic algorithm parameter settings. In file `backend/optimizer/geqo/geqo_params.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

8.4. Further Readings

The following resources contain additional information about genetic algorithms:

- The Hitch-Hiker's Guide to Evolutionary Computation[1] (FAQ for comp.ai.genetic[2])
- Evolutionary Computation and its application to art and design[3] by Craig Reynolds
- [elma99]
- [fong]

[1] http://surf.de.uu.net/encore/www/
[2] news://comp.ai.genetic
[3] http://www.red3d.com/cwr/evolve.html

Chapter 9. Native Language Support

Peter Eisentraut

9.1. For the Translator

PostgreSQL programs (server and client) can issue their messages in your favorite language -- if the messages have been translated. Creating and maintaining translated message sets needs the help of people who speak their own language well and want to contribute to the PostgreSQL effort. You do not have to be a programmer at all to do this. This section explains how to help.

9.1.1. Requirements

We won't judge your language skills -- this section is about software tools. Theoretically, you only need a text editor. But this is only in the unlikely event that you do not want to try out your translated messages. When you configure your source tree, be sure to use the `--enable-nls` option. This will also check for the libintl library and the `msgfmt` program, which all end users will need anyway. To try out your work, follow the applicable portions of the installation instructions.

If you want to start a new translation effort or want to do a message catalog merge (described later), you will need the programs `xgettext` and `msgmerge`, respectively, in a GNU-compatible implementation. Later, we will try to arrange it so that if you use a packaged source distribution, you won't need `xgettext`. (From CVS, you will still need it.) GNU gettext 0.10.36 or later is currently recommended.

Your local gettext implementation should come with its own documentation. Some of that is probably duplicated in what follows, but for additional details you should look there.

9.1.2. Concepts

The pairs of original (English) messages and their (possibly) translated equivalents are kept in *message catalogs*, one for each program (although related programs can share a message catalog) and for each target language. There are two file formats for message catalogs: The first is the “PO” file (for Portable Object), which is a plain text file with special syntax that translators edit. The second is the “MO” file (for Machine Object), which is a binary file generated from the respective PO file and is used while the internationalized program is run. Translators do not deal with MO files; in fact hardly anyone does.

The extension of the message catalog file is to no surprise either `.po` or `.mo`. The base name is either the name of the program it accompanies, or the language the file is for, depending on the situation. This is a bit confusing. Examples are `psql.po` (PO file for psql) or `fr.mo` (MO file in French).

The file format of the PO files is illustrated here:

```
# comment

msgid "original string"
msgstr "translated string"

msgid "more original"
msgstr "another translated"
"string can be broken up like this"

...
```

The msgid's are extracted from the program source. (They need not be, but this is the most common way.) The msgstr lines are initially empty and are filled in with useful strings by the translator. The strings can contain C-style escape characters and can be continued across lines as illustrated. (The next line must start at the beginning of the line.)

The # character introduces a comment. If whitespace immediately follows the # character, then this is a comment maintained by the translator. There may also be automatic comments, which have a non-whitespace character immediately following the #. These are maintained by the various tools that operate on the PO files and are intended to aid the translator.

```
#. automatic comment
#: filename.c:1023
#, flags, flags
```

The #. style comments are extracted from the source file where the message is used. Possibly the programmer has inserted information for the translator, such as about expected alignment. The #: comment indicates the exact location(s) where the message is used in the source. The translator need not look at the program source, but he can if there is doubt about the correct translation. The #, comments contain flags that describe the message in some way. There are currently two flags: `fuzzy` is set if the message has possibly been outdated because of changes in the program source. The translator can then verify this and possibly remove the fuzzy flag. Note that fuzzy messages are not made available to the end user. The other flag is `c-format`, which indicates that the message is a `printf`-style format template. This means that the translation should also be a format string with the same number and type of placeholders. There are tools that can verify this, which key off the c-format flag.

9.1.3. Creating and maintaining message catalogs

Okay, so how does one create a “blank” message catalog? First, go into the directory that contains the program whose messages you want to translate. If there is a file `nls.mk`, then this program has been prepared for translation.

If there are already some `.po` files, then someone has already done some translation work. The files are named *`language`*`.po`, where *`language`* is the ISO 639-1[1] two-letter language code (in lower case), e.g., `fr.po` for French. If there is really a need for more than one translation effort per language then the files may also be named *`language_region`*`.po` where *`region`* is the ISO 3166-1[2] two-letter country code (in upper case), e.g., `pt_BR.po` for Portuguese in Brazil. If you find the language you wanted you can just start working on that file.

If you need to start a new translation effort, then first run the command

```
gmake init-po
```

This will create a file *`progname`*`.pot`. (`.pot` to distinguish it from PO files that are “in production”. What does the T stand for? I don't know.) Copy this file to *`language`*`.po` and edit it. To make it known that the new language is available, also edit the file `nls.mk` and add the language (or language and country) code to the line that looks like:

```
AVAIL_LANGUAGES := de fr
```

(Other languages may appear, of course.)

As the underlying program or library changes, messages may be changed or added by the programmers. In this case you do not need to start from scratch. Instead, run the command

```
gmake update-po
```

[1] http://lcweb.loc.gov/standards/iso639-2/englangn.html
[2] http://www.din.de/gremien/nas/nabd/iso3166ma/codlstp1/en_listp1.html

which will create a new blank message catalog file (the pot file you started with) and will merge it with the existing PO files. If the merge algorithm is not sure about a particular message it marks it "fuzzy" as explained above. For the case where something went really wrong, the old PO file is saved with a `.po.old` extension.

9.1.4. Editing the PO files

The PO files can be edited with a regular text editor. The translator should only change the area between the quotes after the msgstr directive, may add comments and alter the fuzzy flag. There is (unsurprisingly) a PO mode for Emacs, which I find quite useful.

The PO files need not be completely filled in. The software will automatically fall back to the original string if no translation (or an empty translation) is available. It is no problem to submit incomplete translations for inclusions in the source tree; that gives room for other people to pick up your work. However, you are encouraged to give priority to removing fuzzy entries after doing a merge. Remember that fuzzy entries will not be installed; they only serve as reference what might be the right translation.

Here are some things to keep in mind while editing the translations:

- Make sure that if the original ends with a newline, the translation does, too. Similarly for tabs, etc.
- If the original is a printf format string, the translation also needs to be. The translation also needs to have the same format specifiers in the same order. Sometimes the natural rules of the language make this impossible or at least awkward. In this case you can use this format:

  ```
  msgstr "Die Datei %2$s hat %1$u Zeichen."
  ```

  Then the first placeholder will actually use the second argument from the list. The *`digits`*`$` needs to follow the % and come before any other format manipulators. (This feature really exists in the `printf` family of functions. You may not have heard of it because there is little use for it outside of message internationalization.)
- If the original string contains a linguistic mistake, report that (or fix it yourself in the program source) and translate normally. The corrected string can be merged in when the program sources have been updated. If the original string contains a factual mistake, report that (or fix it yourself) and do not translate it. Instead, you may mark the string with a comment in the PO file.
- Maintain the style and tone of the original string. Specifically, messages that are not sentences (`cannot open file %s`) should probably not start with a capital letter (if your language distinguishes letter case) or end with a period (if your language uses punctuation marks).
- If you don't know what a message means, or if it is ambiguous, ask on the developers' mailing list. Chances are that English speaking end users might also not understand it or find it ambiguous, so it's best to improve the message.

9.2. For the Programmer

This section describes how to support native language support in a program or library that is part of the PostgreSQL distribution. Currently, it only applies to C programs.

Adding NLS support to a program

1. Insert this code into the startup sequence of the program:

   ```
   #ifdef ENABLE_NLS
   #include <locale.h>
   ```

```
#endif

...

#ifdef ENABLE_NLS
setlocale(LC_ALL, "");
bindtextdomain("progname", LOCALEDIR);
textdomain("progname");
#endif
```

(The *progname* can actually be chosen freely.)

2. Wherever a message that is a candidate for translation is found, a call to `gettext()` needs to be inserted. E.g.,

   ```
   fprintf(stderr, "panic level %d\n", lvl);
   ```

   would be changed to

   ```
   fprintf(stderr, gettext("panic level %d\n"), lvl);
   ```

   (`gettext` is defined as a no-op if no NLS is configured.)

   This may tend to add a lot of clutter. One common shortcut is to

   ```
   #define _(x) gettext((x))
   ```

   Another solution is feasible if the program does much of its communication through one or a few functions, such as `elog()` in the backend. Then you make this function call `gettext` internally on all input values.

3. Add a file `nls.mk` in the directory with the program sources. This file will be read as a makefile. The following variable assignments need to be made here:

   CATALOG_NAME

   The program name, as provided in the `textdomain()` call.

   AVAIL_LANGUAGES

   List of provided translations -- empty in the beginning.

   GETTEXT_FILES

   List of files that contain translatable strings, i.e., those marked with `gettext` or an alternative solution. Eventually, this will include nearly all source files of the program. If this list gets too long you can make the first "file" be a + and the second word be a file that contains one file name per line.

   GETTEXT_TRIGGERS

   The tools that generate message catalogs for the translators to work on need to know what function calls contain translatable strings. By default, only `gettext()` calls are known. If you used _ or other identifiers you need to list them here. If the translatable string is not the first argument, the item needs to be of the form `func:2` (for the second argument).

The build system will automatically take care of building and installing the message catalogs.

To ease the translation of messages, here are some guidelines:

- Do not construct sentences at run-time out of laziness, like

  ```
  printf("Files where %s.\n", flag ? "copied" : "removed");
  ```

  The word order within the sentence may be different in other languages.

- For similar reasons, this won't work:

  ```
  printf("copied %d file%s", n, n!=1 ? "s" : "");
  ```

  because it assumes how the plural is formed. If you figured you could solve it like this

  ```
  if (n==1)
      printf("copied 1 file");
  else
      printf("copied %d files", n):
  ```

  then be disappointed. Some languages have more than two forms, with some peculiar rules. We may have a solution for this in the future, but for now this is best avoided altogether. You could write:

  ```
  printf("number of copied files: %d", n);
  ```

- If you want to communicate something to the translator, such as about how a message is intended to line up with other output, precede the occurrance of the string with a comment that starts with `translator`, e.g.,

  ```
  /* translator: This message is not what it seems to be. */
  ```

  These comments are copied to the message catalog files so that the translators can see them.

Appendix A. The CVS Repository

Marc Fournier
Tom Lane
Thomas Lockhart

The PostgreSQL source code is stored and managed using the CVS code management system.

At least two methods, anonymous CVS and CVSup, are available to pull the CVS code tree from the PostgreSQL server to your local machine.

A.1. Getting The Source Via Anonymous CVS

If you would like to keep up with the current sources on a regular basis, you can fetch them from our CVS server and then use CVS to retrieve updates from time to time.

Anonymous CVS

1. You will need a local copy of CVS (Concurrent Version Control System), which you can get from http://www.cyclic.com/ or any GNU software archive site. We currently recommend version 1.10 (the most recent at the time of writing). Many systems have a recent version of cvs installed by default.

2. Do an initial login to the CVS server:

   ```
   $ cvs -d :pserver:anoncvs@anoncvs.postgresql.org:/projects/cvsroot
    login
   ```

   You will be prompted for a password; just press `ENTER`. You should only need to do this once, since the password will be saved in `.cvspass` in your home directory.

3. Fetch the PostgreSQL sources:

   ```
   cvs -z3 -d :pserver:anoncvs@anoncvs.postgresql.org:/projects/
   cvsroot co -P pgsql
   ```

   which installs the PostgreSQL sources into a subdirectory `pgsql` of the directory you are currently in.

   #### Note

   If you have a fast link to the Internet, you may not need `-z3`, which instructs CVS to use gzip compression for transferred data. But on a modem-speed link, it's a very substantial win.

   This initial checkout is a little slower than simply downloading a `tar.gz` file; expect it to take 40 minutes or so if you have a 28.8K modem. The advantage of CVS doesn't show up until you want to update the file set later on.

4. Whenever you want to update to the latest CVS sources, `cd` into the `pgsql` subdirectory, and issue

   ```
   $ cvs -z3 update -d -P
   ```

   This will fetch only the changes since the last time you updated. You can update in just a couple of minutes, typically, even over a modem-speed line.

5. You can save yourself some typing by making a file `.cvsrc` in your home directory that contains

   ```
   cvs -z3
   update -d -P
   ```

   This supplies the `-z3` option to all cvs commands, and the `-d` and `-P` options to cvs update. Then you just have to say

   ```
   $ cvs update
   ```

   to update your files.

   ### Caution

   Some older versions of CVS have a bug that causes all checked-out files to be stored world-writable in your directory. If you see that this has happened, you can do something like

   ```
   $ chmod -R go-w pgsql
   ```

   to set the permissions properly. This bug is fixed as of CVS version 1.9.28.

CVS can do a lot of other things, such as fetching prior revisions of the PostgreSQL sources rather than the latest development version. For more info consult the manual that comes with CVS, or see the online documentation at http://www.cyclic.com/.

A.2. CVS Tree Organization

Author

Written by Marc G. Fournier (<scrappy@hub.org>) on 1998-11-05

The command `cvs checkout` has a flag, `-r`, that lets you check out a certain revision of a module. This flag makes it easy to, for example, retrieve the sources that make up release 6_4 of the module `tc' at any time in the future:

```
$ cvs checkout -r REL6_4 tc
```

This is useful, for instance, if someone claims that there is a bug in that release, but you cannot find the bug in the current working copy.

Tip

You can also check out a module as it was at any given date using the `-D` option.

When you tag more than one file with the same tag you can think about the tag as “a curve drawn through a matrix of filename vs. revision number”. Say we have 5 files with the following revisions:

```
file1   file2   file3   file4   file5

1.1     1.1     1.1     1.1  /--1.1*      <-*-  TAG
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
```

then the tag `TAG` will reference file1-1.2, file2-1.3, etc.

Note

For creating a release branch, other then a -b option added to the command, it's the same thing.

So, to create the 6.4 release I did the following:

```
$ cd pgsql
$ cvs tag -b REL6_4
```

which will create the tag and the branch for the RELEASE tree.

For those with CVS access, it's simple to create directories for different versions. First, create two subdirectories, RELEASE and CURRENT, so that you don't mix up the two. Then do:

```
cd RELEASE
cvs checkout -P -r REL6_4 pgsql
cd ../CURRENT
cvs checkout -P pgsql
```

which results in two directory trees, `RELEASE/pgsql` and `CURRENT/pgsql`. From that point on, CVS will keep track of which repository branch is in which directory tree, and will allow independent updates of either tree.

If you are *only* working on the `CURRENT` source tree, you just do everything as before we started tagging release branches.

After you've done the initial checkout on a branch

```
$ cvs checkout -r REL6_4
```

anything you do within that directory structure is restricted to that branch. If you apply a patch to that directory structure and do a

```
cvs commit
```

while inside of it, the patch is applied to the branch and *only* the branch.

A.3. Getting The Source Via CVSup

An alternative to using anonymous CVS for retrieving the PostgreSQL source tree is CVSup. CVSup was developed by John Polstra (<jdp@polstra.com>) to distribute CVS repositories and other file trees for the FreeBSD project[1].

[1] http://www.freebsd.org

A major advantage to using CVSup is that it can reliably replicate the *entire* CVS repository on your local system, allowing fast local access to cvs operations such as `log` and `diff`. Other advantages include fast synchronization to the PostgreSQL server due to an efficient streaming transfer protocol which only sends the changes since the last update.

A.3.1. Preparing A CVSup Client System

Two directory areas are required for CVSup to do it's job: a local CVS repository (or simply a directory area if you are fetching a snapshot rather than a repository; see below) and a local CVSup bookkeeping area. These can coexist in the same directory tree.

Decide where you want to keep your local copy of the CVS repository. On one of our systems we recently set up a repository in `/home/cvs/`, but had formerly kept it under a PostgreSQL development tree in `/opt/postgres/cvs/`. If you intend to keep your repository in `/home/cvs/`, then put

```
setenv CVSROOT /home/cvs
```

in your `.cshrc` file, or a similar line in your `.bashrc` or `.profile` file, depending on your shell.

The cvs repository area must be initialized. Once `CVSROOT` is set, then this can be done with a single command:

```
$ cvs init
```

after which you should see at least a directory named `CVSROOT` when listing the `CVSROOT` directory:

```
$ ls $CVSROOT
CVSROOT/
```

A.3.2. Running a CVSup Client

Verify that cvsup is in your path; on most systems you can do this by typing

```
which cvsup
```

Then, simply run cvsup using:

```
$ cvsup -L 2 postgres.cvsup
```

where `-L 2` enables some status messages so you can monitor the progress of the update, and *`postgres.cvsup`* is the path and name you have given to your CVSup configuration file.

Here is a CVSup configuration file modified for a specific installation, and which maintains a full local CVS repository:

```
# 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
```

The following is a suggested CVSup config file from the PostgreSQL ftp site[2] which will fetch the current snapshot only:

```
# 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
```

[2] ftp://ftp.postgresql.org/pub/CVSup/README.cvsup

```
# individual distributions vs 'the whole thing'
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

A.3.3. Installing CVSup

CVSup is available as source, pre-built binaries, or Linux RPMs. It is far easier to use a binary than to build from source, primarily because the very capable, but voluminous, Modula-3 compiler is required for the build.

CVSup Installation from Binaries

You can use pre-built binaries if you have a platform for which binaries are posted on the PostgreSQL ftp site[3], or if you are running FreeBSD, for which CVSup is available as a port.

> **Note**
>
> CVSup was originally developed as a tool for distributing the FreeBSD source tree. It is available as a "port", and for those running FreeBSD, if this is not sufficient to tell how to obtain and install it then please contribute a procedure here.

At the time of writing, binaries are available for Alpha/Tru64, ix86/xBSD, HPPA/HP-UX 10.20, MIPS/IRIX, ix86/linux-libc5, ix86/linux-glibc, Sparc/Solaris, and Sparc/SunOS.

1. Retrieve the binary tar file for cvsup (cvsupd is not required to be a client) appropriate for your platform.

   a. (Optional) If you are running FreeBSD, install the CVSup port.

   b. (Optional) If you have another platform, check for and download the appropriate binary from the PostgreSQL ftp site[4].

2. Check the tar file to verify the contents and directory structure, if any. For the linux tar file at least, the static binary and man page is included without any directory packaging.

   a. If the binary is in the top level of the tar file, then simply unpack the tar file into your target directory:

      ```
      $ cd /usr/local/bin
      $ tar zxvf /usr/local/src/cvsup-16.0-linux-i386.tar.gz
      $ mv cvsup.1 ../doc/man/man1/
      ```

   b. If there is a directory structure in the tar file, then unpack the tar file within /usr/local/src and move the binaries into the appropriate location as above.

3. Ensure that the new binaries are in your path.

   ```
   $ rehash
   ```

[3] ftp://ftp.postgresql.org/pub
[4] ftp://ftp.postgresql.org/pub

```
$ which cvsup
$ set path=(path to cvsup $path)
$ which cvsup
/usr/local/bin/cvsup
```

A.3.4. Installation from Sources

Installing CVSup from sources is not entirely trivial, primarily because most systems will need to install a Modula-3 compiler first. This compiler is available as Linux RPM, FreeBSD package, or source code.

Note

A clean-source installation of Modula-3 takes roughly 200MB of disk space, which shrinks to roughly 50MB of space when the sources are removed.

Linux installation

1. Install Modula-3.

   a. Pick up the Modula-3 distribution from Polytechnique Montréal[5], who are actively maintaining the code base originally developed by the DEC Systems Research Center[6]. The PM3 RPM distribution is roughly 30MB compressed. At the time of writing, the 1.1.10-1 release installed cleanly on RH-5.2, whereas the 1.1.11-1 release is apparently built for another release (RH-6.0?) and does not run on RH-5.2.

   ## Tip

   This particular rpm packaging has *many* RPM files, so you will likely want to place them into a separate directory.

   b. Install the Modula-3 rpms:

   ```
   # rpm -Uvh pm3*.rpm
   ```

2. Unpack the cvsup distribution:

   ```
   # cd /usr/local/src
   # tar zxf cvsup-16.0.tar.gz
   ```

3. Build the cvsup distribution, suppressing the GUI interface feature to avoid requiring X11 libraries:

   ```
   # make M3FLAGS="-DNOGUI"
   ```

   and if you want to build a static binary to move to systems that may not have Modula-3 installed, try:

   ```
   # make M3FLAGS="-DNOGUI -DSTATIC"
   ```

4. Install the built binary:

[5] http://m3.polymtl.ca/m3
[6] http://www.research.digital.com/SRC/modula-3/html/home.html

```
# make M3FLAGS="-DNOGUI -DSTATIC" install
```

Appendix B. Documentation

PostgreSQL has four primary documentation formats:

- Plain text, for pre-installation information
- HTML, for on-line browsing and reference
- Postscript, for printing
- man pages, for quick reference.

Additionally, a number of plain-text README-type files can be found throughout the PostgreSQL source tree, documenting various implementation issues.

The documentation is organized into several “books”:

- *Tutorial*: introduction for new users
- *User's Guide*: documents the SQL implementation
- *Reference Manual*: reference pages for programs and SQL commands
- *Administrator's Guide*: installation and server maintenance
- *Programmer's Guide*: programming client applications and server extensions
- *Developer's Guide*: assorted information for developers of PostgreSQL proper

All books are available as HTML and Postscript. The *Reference Manual* contains reference entries which are also shipped as man pages.

HTML documentation and man pages are part of a standard distribution and are installed by default. Postscript format documentation is available separately for download.

B.1. DocBook

The documentation sources are written in *DocBook*, which is a markup language superficially similar to HTML. Both of these languages are applications of the *Standard Generalized Markup Language*, SGML, which is essentially a language for describing other languages. In what follows, the terms DocBook and SGML are both used, but technically they are not interchangeable.

DocBook allows an author to specify the structure and content of a technical document without worrying about presentation details. A document style defines how that content is rendered into one of several final forms. DocBook is maintained by the OASIS[1] group. The official DocBook site[2] has good introductory and reference documentation and a complete O'Reilly book for your online reading pleasure. The FreeBSD Documentation Project[3] also uses DocBook and has some good information, including a number of style guidelines that might be worth considering.

B.2. Toolsets

The following tools are used to process the documentation. Some may be optional, as noted.

[1] http://www.oasis-open.org
[2] http://www.oasis-open.org/docbook
[3] http://www.freebsd.org/docproj/docproj.html

DocBook DTD[4]

This is the definition of DocBook itself. We currently use version 3.1; you cannot use later or earlier versions. Note that there is also an XML version of DocBook -- do not use that.

ISO 8879 character entities[5]

These are required by DocBook but are distributed separately because they are maintained by ISO.

OpenJade[6]

This is the base package of SGML processing. It contains an SGML parser, a DSSSL processor (that is, a program to convert SGML to other formats using DSSSL stylesheets), as well as a number of related tools. Jade is now being maintained by the OpenJade group, no longer by James Clark.

DocBook DSSSL Stylesheets[7]

These contain the processing instructions for converting the DocBook sources to other formats, such as HTML.

DocBook2X tools[8]

This optional package is used to create man pages. It has a number of prerequisite packages of its own. Check the web site.

JadeTeX[9]

If you want to, you can also install JadeTeX to use TeX as a formatting backend for Jade. JadeTeX can create Postscript or PDF files (the latter with bookmarks).

However, the output from JadeTeX is inferior to what you get from the RTF backend. Particular problem areas are tables and various artifacts of vertical and horizontal spacing. Also, there is no opportunity to manually polish the results.

We have documented experience with several installation methods for the various tools that are needed to process the documentation. These will be described below. There may be some other packaged distributions for these tools. Please report package status to the docs mailing list and we will include that information here.

B.2.1. Linux RPM Installation

Many vendors provide a complete RPM set for DocBook processing in their distribution, which is usually based on the docbook-tools[10] effort at Red Hat Software. Look for an “SGML” option while installing, or the following packages: `sgml-common`, `docbook`, `stylesheets`, `openjade` (or `jade`). Possibly `sgml-tools` will be needed as well. If your distributor does not provide these then you should be able to make use of the packages from some other, reasonably compatible vendor.

[4] http://www.oasis-open.org/docbook/sgml/
[5] http://www.oasis-open.org/cover/ISOEnts.zip
[6] http://openjade.sourceforge.net
[7] http://docbook.sourceforge.net/projects/dsssl/index.html
[8] http://docbook2x.sourceforge.net
[9] http://jadetex.sourceforge.net
[10] http://sources.redhat.com/docbook-tools/

B.2.2. FreeBSD Installation

The FreeBSD Documentation Project is itself a heavy user of DocBook, so it comes as no surprise that there is a full set of "ports" of the documentation tools available on FreeBSD. The following ports need to be installed to build the documentation on FreeBSD.

- `textproc/sp`
- `textproc/openjade`
- `textproc/docbook-310`
- `textproc/iso8879`
- `textproc/dsssl-docbook-modular`

A number of things from `/usr/ports/print` (`tex`, `jadetex`) might also be of interest.

It's possible that the ports do not update the main catalog file in `/usr/local/share/sgml/catalog`. Be sure to have the following line in there:

```
CATALOG "/usr/local/share/sgml/docbook/3.1/catalog"
```

If you do not want to edit the file you can also set the environment variable `SGML_CATALOG_FILES` to a colon-separated list of catalog files (such as the one above).

More information about the FreeBSD documentation tools can be found in the FreeBSD Documentation Project's instructions[11].

B.2.3. Debian Packages

There is a full set of packages of the documentation tools available for Debian GNU/Linux. To install, simply use:

```
apt-get install jade
apt-get install docbook
apt-get install docbook-stylesheets
```

B.2.4. Manual Installation from Source

The manual installation process of the DocBook tools is somewhat complex, so if you have pre-built packages available, use them. We describe here only a standard setup, with reasonabley standard installation paths, and no "fancy" features. For details, you should study the documentation of the respective package, and read SGML introductory material.

B.2.4.1. Installing OpenJade

1. The installation of OpenJade offers a GNU-style `./configure; make; make install` build process. Details can be found in the OpenJade source distribution. In a nutshell:

   ```
   ./configure --enable-default-catalog=/usr/local/share/sgml/catalog
   ```

[11] http://www.freebsd.org/doc/en_US.ISO8859-1/books/fdp-primer/tools.html

```
make
make install
```

Be sure to remember where you put the "default catalog"; you will need it below. You can also leave it off, but then you will have to set the environment variable `SGML_CATALOG_FILES` to point to the file whenever you use jade later on. (This method is also an option if OpenJade is already installed and you want to install the rest of the toolchain locally.)

2. Additionally, you should install the files `dsssl.dtd`, `fot.dtd`, `style-sheet.dtd`, and `catalog` from the `dsssl` directory somewhere, perhaps into `/usr/local/share/sgml/dsssl`. It's probably easiest to copy the entire directory:

   ```
   cp -R dsssl /usr/local/share/sgml
   ```

3. Finally, create the file `/usr/local/share/sgml/catalog` and add this line to it:

   ```
   CATALOG "dsssl/catalog"
   ```

   (This is a relative path reference to the file installed in Step 2. Be sure to adjust it if you chose your installation layout differently.)

B.2.4.2. Installing the DocBook DTD Kit

1. Obtain the DocBook V3.1[12] distribution.

2. Create the directory `/usr/local/share/sgml/docbook31` and change to it. (The exact location is irrelevant, but this one is reasonable within the layout we are following here.)

   ```
   $ mkdir /usr/local/share/sgml/docbook31
   $ cd /usr/local/share/sgml/docbook31
   ```

3. Unpack the archive.

   ```
   $ unzip -a ...../docbk31.zip
   ```

   (The archive will unpack its files into the current directory.)

4. Edit the file `/usr/local/share/sgml/catalog` (or whatever you told jade during installation) and put a line like this into it:

   ```
   CATALOG "docbook31/docbook.cat"
   ```

5. Optionally, you can edit the file `docbook.cat` and comment out or remove the line containing `DTDDECL`. If you do not then you will get warnings from jade, but there is no further harm.

6. Download the ISO 8879 character entities[13] archive, unpack it, and put the files in the same directory you put the DocBook files in.

   ```
   $ cd /usr/local/share/sgml/docbook31
   $ unzip ...../ISOEnts.zip
   ```

7. Run the following command in the directory with the DocBook and ISO files:

   ```
   perl -pi -e 's/iso-(.*).gml/ISO\1/g' docbook.cat
   ```

[12] http://www.oasis-open.org/docbook/sgml/3.1/docbk31.zip
[13] http://www.oasis-open.org/cover/ISOEnts.zip

(This fixes a mixup between the names used in the DocBook catalog file and the actual names of the ISO character entity files.)

B.2.4.3. Installing the DocBook DSSSL Style Sheets

To install the style sheets, unzip and untar the distribution and move it to a suitable place, for example `/usr/local/share/sgml`. (The archive will automatically create a subdirectory.)

```
$ gunzip docbook-dsssl-1.xx.tar.gz
$ tar -C /usr/local/share/sgml -xf docbook-dsssl-1.xx.tar
```

The usual catalog entry in `/usr/local/share/sgml/catalog` can also be made:

```
CATALOG "docbook-dsssl--1.xx/catalog
```

Because stylesheets change rather often, and it's sometimes beneficial to try out alternative versions, PostgreSQL doesn't use this catalog entry. See Section B.3, “Building The Documentation” for information about how to select the stylesheets instead.

B.2.4.4. Installing JadeTeX

To install and use JadeTeX, you will need a working installation of TeX and LaTeX2e, including the supported tools and graphics packages, Babel, AMS fonts and AMS-LaTeX, the PSNFSS extension and companion kit of “the 35 fonts”, the dvips program for generating PostScript, the macro packages fancyhdr, hyperref, minitoc, url and ot2enc. All of these can be found on your friendly neighborhood CTAN[14] site. The installation of the TeX base system is far beyond the scope of this introduction. Binary packages should be available for any system that can run TeX.

Before you can use JadeTeX with the PostgreSQL documentation sources, you will need to increase the size of TeX's internal data structures. Details on this can be found in the JadeTeX installation instructions.

Once that is finished you can install JadeTeX:

```
$ gunzip jadetex-xxx.tar.gz
$ tar xf jadetex-xxx.tar
$ cd jadetex
$ make install
$ mktexlsr
```

The last two need to be done as root.

B.3. Building The Documentation

Before you can build the documentation you need to run the `configure` script as you would when building the programs themselves. Check the output near the end of the run, it should look something like this:

```
checking for onsgmls... onsgmls
checking for openjade... openjade
checking for DocBook V3.1... yes
```

[14] http://www.ctan.org

```
checking for DocBook stylesheets... /usr/lib/sgml/stylesheets/nwalsh-
modular
checking for sgmlspl... sgmlspl
```

If neither `onsgmls` nor `nsgmls` were found then you will not see the remaining 4 lines. `nsgmls` is part of the Jade package. If "DocBook V3.1" was not found then you did not install the DocBook DTD kit in a place where jade can find it, or you have not set up the catalog files correctly. See the installation hints above. The DocBook stylesheets are looked for in a number of relatively standard places, but if you have them some other place then you should set the environment variable `DOCBOOKSTYLE` to the location and rerun `configure` afterwards.

Once you have everything set up, change to the directory `doc/src/sgml` and run one of the following commands: (Remember to use GNU make.)

- To build the HTML version of the *Administrator's Guide*:

  ```
  doc/src/sgml$ gmake admin.html
  ```

- For the RTF version of the same:

  ```
  doc/src/sgml$ gmake admin.rtf
  ```

- To get a DVI version via JadeTeX:

  ```
  doc/src/sgml$ gmake admin.dvi
  ```

- And Postscript from the DVI:

  ```
  doc/src/sgml$ gmake admin.ps
  ```

  ### Note

  The official Postscript format documentation is generated differently. See Section B.3.3, "Hardcopy Generation" below.

The other books can be built with analogous commands by replacing `admin` with one of `developer`, `programmer`, `tutorial`, or `user`. Using `postgres` builds an integrated version of all 5 books, which is practical since the browser interface makes it easy to move around all of the documentation by just clicking.

B.3.1. HTML

When building HTML documentation in `doc/src/sgml`, some of the resulting files will possibly (or quite certainly) have conflicting names between books. Therefore the files are not in that directory in the regular distribution. Instead, the files belonging to each book are stored in a tar archive that is unpacked at installation time. To create a set of HTML documentation packages use the commands

```
cd doc/src
gmake tutorial.tar.gz
gmake user.tar.gz
gmake admin.tar.gz
gmake programmer.tar.gz
gmake postgres.tar.gz
gmake install
```

In the distribution, these archives live in the `doc` directory and are installed by default with `gmake install`.

B.3.2. Manpages

We use the docbook2man utility to convert DocBook `REFENTRY` pages to *roff output suitable for man pages. The man pages are also distributed as a tar archive, similar to the HTML version. To create the man page package, use the commands

```
cd doc/src
gmake man
```

which will result in a tar file being generated in the `doc/src` directory.

The man build leaves a lot of confusing output, and special care must be taken to produce quality results. There is still room for improvement in this area.

B.3.3. Hardcopy Generation

The hardcopy Postscript documentation is generated by converting the SGML source code to RTF, then importing into ApplixWare. After a little cleanup (see the following section) the output is “printed” to a postscript file.

Several areas are addressed while generating Postscript hardcopy, including RTF repair, ToC generation, and page break adjustments.

Applixware RTF Cleanup

jade, an integral part of the hardcopy procedure, omits specifying a default style for body text. In the past, this undiagnosed problem led to a long process of Table of Contents (ToC) generation. However, with great help from the ApplixWare folks the symptom was diagnosed and a workaround is available.

1. Generate the RTF input by typing (for example):

   ```
   % cd doc/src/sgml
   % make tutorial.rtf
   ```

2. Repair the RTF file to correctly specify all styles, in particular the default style. If the document contains `REFENTRY` sections, one must also replace formatting hints which tie a *preceding* paragraph to the current paragraph, and instead tie the current paragraph to the following one. A utility, fixrtf is available in `doc/src/sgml` to accomplish these repairs:

   ```
   % cd doc/src/sgml
   % fixrtf tutorial.rtf
   ```

   or

   ```
   % cd doc/src/sgml
   % fixrtf --refentry reference.rtf
   ```

   The script adds `{\s0 Normal;}` as the zero-th style in the document. According to ApplixWare, the RTF standard would prohibit adding an implicit zero-th style, though M$Word happens to handle this case. For repairing `REFENTRY` sections, the script replaces `\keepn` tags with `\keep`.

3. Open a new document in Applix Words and then import the RTF file.

4. Generate a new ToC using ApplixWare.

   a. Select the existing ToC lines, from the beginning of the first character on the first line to the last character of the last line.

   b. Build a new ToC using `Tools.BookBuilding.CreateToC`. Select the first three levels of headers for inclusion in the ToC. This will replace the existing lines imported in the RTF with a native ApplixWare ToC.

   c. Adjust the ToC formatting by using `Format.Style`, selecting each of the three ToC styles, and adjusting the indents for `First` and `Left`. Use the following values:

Table B.1. Indent Formatting for Table of Contents

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. Work through the document to:

   - Adjust page breaks.
   - Adjust table column widths.
   - Insert figures into the document. Center each figure on the page using the centering margins button on the ApplixWare toolbar.

> **Note**
>
> Not all documents have figures. You can grep the SGML source files for the string `graphic` to identify those parts of the documentation that may have figures. A few figures are replicated in various parts of the documentation.

6. Replace the right-justified page numbers in the Examples and Figures portions of the ToC with correct values. This only takes a few minutes per document.

7. (Optional) Delete the index section from the document if it is empty.

8. Regenerate and adjust the table of contents.

   a. Select the ToC field.

   b. Select `Tools->Book Building->Create Table of Contents`.

   c. Unbind the ToC by selecting `Tools->Field Editing->Unprotect`.

   d. Delete the first line in the ToC, which is an entry for the ToC itself.

9. Save the document as native Applix Words format to allow easier last minute editing later.

10. “Print” the document to a file in Postscript format.

11. Compress the Postscript file using gzip. Place the compressed file into the `doc` directory.

B.3.4. Plain Text Files

Several files are distributed as plain text, for reading during the installation process. The `INSTALL` file corresponds to the chapter in the *Administrator's Guide*, with some minor changes to account for the different context. To recreate the file, change to the directory `doc/src/sgml` and enter **`gmake INSTALL`**. This will create a file `INSTALL.html` that can be saved as text with Netscape Navigator and put into the place of the existing file. Netscape seems to offer the best quality for HTML to text conversions (over lynx and w3m).

The file `HISTORY` can be created similarly, using the command **`gmake HISTORY`**. For the file `src/test/regress/README` the command is **`gmake regress_README`**.

B.4. Documentation Authoring

SGML and DocBook do not suffer from an oversupply of open-source authoring tools. The most common toolset is the Emacs/XEmacs editor with appropriate editing mode. On some systems these tools are provided in a typical full installation.

B.4.1. Emacs/PSGML

PSGML is the most common and most powerful mode for editing SGML documents. When properly configured, it will allow you to use Emacs to insert tags and check markup consistency. You could use it for HTML as well. Check the PSGML web site[15] for downloads, installation instructions, and detailed documentation.

There is one important thing to note with PSGML: its author assumed that your main SGML DTD directory would be `/usr/local/lib/sgml`. If, as in the examples in this chapter, you use `/usr/local/share/sgml`, you have to compensate for this, either by setting `SGML_CATALOG_FILES` environment variable, or you can customize your PSGML installation (its manual tells you how).

Put the following in your `~/.emacs` environment file (adjusting the path names to be appropriate for your system):

```
; ********** 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 )
```

and in the same file add an entry for SGML into the (existing) definition for `auto-mode-alist`:

```
(setq
```

[15] http://www.lysator.liu.se/projects/about_psgml.html

```
  auto-mode-alist
  '(("\\.sgml$" . sgml-mode)
   ))
```

Currently, each SGML source file has the following block at the end of the file:

```
<!-- 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:
-->
```

This will set up a number of editing mode parameters even if you do not set up your `~/.emacs` file, but it is a bit unfortunate, since if you followed the installation instructions above, then the catalog path will not match your location. Hence you might need to turn off local variables:

```
(setq inhibit-local-variables t)
```

The PostgreSQL distribution includes a parsed DTD definitions file `reference.ced`. You may find that when using PSGML, a comfortable way of working with these separate files of book parts is to insert a proper `DOCTYPE` declaration while you're editing them. If you are working on this source, for instance, it is an appendix chapter, so you would specify the document as an “appendix” instance of a DocBook document by making the first line look like this:

```
<!doctype appendix PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
```

This means that anything and everything that reads SGML will get it right, and I can verify the document with `nsgmls -s docguide.sgml`. (But you need to take out that line before building the entire documentation set.)

B.4.2. Other Emacs modes

GNU Emacs ships with a different SGML mode, which is not quite as powerful as PSGML, but it's less confusing and lighter weight. Also, it offers syntax highlighting (font lock), which can be very helpful.

Norm Walsh offers a major mode specifically for DocBook[16] which also has font-lock and a number of features to reduce typing.

[16] http://nwalsh.com/emacs/docbookide/index.html