

# **PostgreSQL Programmer's Guide**

**The PostgreSQL Development Team**

Edited by  
**Thomas Lockhart**

**PostgreSQL Programmer's Guide**  
by The PostgreSQL Development Team

Edited by Thomas Lockhart

PostgreSQL  
is Copyright © 1996-2000 by PostgreSQL Inc.

# Table of Contents

Table of Contents	i
List of Tables	ix
List of Figures	x
List of Examples	xi
Summary	i
Chapter 1. Introduction	1
Resources	1
Terminology	2
Notation	3
Problem Reporting Guidelines	3
Identifying Bugs	3
What to report	4
Where to report bugs	6
Y2K Statement	6
Copyrights and Trademarks	7
Chapter 2. Architecture	8
Postgres Architectural Concepts	8
Chapter 3. Extending SQL: An Overview	11
How Extensibility Works	11
The Postgres Type System	11
About the Postgres System Catalogs	12
Chapter 4. Extending SQL: Functions	15
Query Language (SQL) Functions	15
Examples	15
SQL Functions on Base Types	16
SQL Functions on Composite Types	16
Procedural Language Functions	19
Internal Functions	19
Compiled (C) Language Functions	19
C Language Functions on Base Types	20
C Language Functions on Composite Types	24
Writing Code	25
Function Overloading	26
Name Space Conflicts	26
Chapter 5. Extending SQL: Types	27
User-Defined Types	27
Functions Needed for a User-Defined Type	27
Large Objects	28
Chapter 6. Extending SQL: Operators	29
Operator Optimization Information	30
COMMUTATOR	30
NEGATOR	31
RESTRICT	31
JOIN	32
HASHES	32
SORT1 and SORT2	33
Chapter 7. Extending SQL: Aggregates	35
Chapter 8. The Postgres Rule System	37
What is a Querytree?	37
The Parts of a Querytree	37
Views and the Rule System	39
Implementation of Views in Postgres	39
How SELECT Rules Work	39
View Rules in Non-SELECT Statements	46
The Power of Views in Postgres	47
Implementation Side Effects	48

Rules on INSERT, UPDATE and DELETE . . . . .	48
Differences to View Rules . . . . .	48
How These Rules Work . . . . .	49
Cooperation with Views . . . . .	53
Rules and Permissions . . . . .	59
Rules versus Triggers . . . . .	60
Chapter 9. Interfacing Extensions To Indices . . . . .	64
Chapter 10. Index Cost Estimation Functions . . . . .	70
Chapter 11. GiST Indices . . . . .	73
Chapter 12. Procedural Languages . . . . .	75
Installing Procedural Languages . . . . .	75
Chapter 13. Linking Dynamically-Loaded Functions . . . . .	77
Linux . . . . .	78
DEC OSF/1 . . . . .	78
SunOS 4.x, Solaris 2.x and HP-UX . . . . .	78
Chapter 14. Triggers . . . . .	80
Trigger Creation . . . . .	80
Interaction with the Trigger Manager . . . . .	81
Visibility of Data Changes . . . . .	83
Examples . . . . .	83
Chapter 15. Server Programming Interface . . . . .	87
Interface Functions . . . . .	88
SPI_connect . . . . .	88
Name . . . . .	88
Synopsis . . . . .	88
Description . . . . .	88
Usage . . . . .	88
Algorithm . . . . .	88
SPI_finish . . . . .	89
Name . . . . .	89
Synopsis . . . . .	89
Description . . . . .	89
Usage . . . . .	89
Algorithm . . . . .	89
SPI_exec . . . . .	90
Name . . . . .	90
Synopsis . . . . .	90
Description . . . . .	91
Usage . . . . .	91
Algorithm . . . . .	91
SPI_prepare . . . . .	92
Name . . . . .	92
Synopsis . . . . .	92
Description . . . . .	92
Usage . . . . .	92
SPI_saveplan . . . . .	93
Name . . . . .	93
Synopsis . . . . .	93
Description . . . . .	93
Usage . . . . .	93
SPI_execp . . . . .	94
Name . . . . .	94
Synopsis . . . . .	94
Description . . . . .	94
Usage . . . . .	95
Interface Support Functions . . . . .	95
SPI_copytuple . . . . .	95
Name . . . . .	95
Synopsis . . . . .	95
Description . . . . .	95

Usage	95
SPI_modifytuple	96
Name	96
Synopsis	96
Description	97
Usage	97
SPI_fnumber	97
Name	97
Synopsis	97
Description	97
Usage	97
SPI_fname	98
Name	98
Synopsis	98
Description	98
Usage	98
Algorithm	98
SPI_getvalue	99
Name	99
Synopsis	99
Description	99
Usage	99
Algorithm	99
SPI_getbinval	100
Name	100
Synopsis	100
Description	100
Usage	100
Algorithm	100
SPI_gettype	101
Name	101
Synopsis	101
Description	101
Usage	101
Algorithm	101
SPI_gettypeid	102
Name	102
Synopsis	102
Description	102
Usage	102
Algorithm	102
SPI_getrelname	103
Name	103
Synopsis	103
Description	103
Usage	103
Algorithm	103
SPI_palloc	104
Name	104
Synopsis	104
Description	104
Usage	104
SPI_realloc	105
Name	105
Synopsis	105
Description	105
Usage	105
SPI_pfree	106
Name	106
Synopsis	106

Description . . . . .	106
Usage . . . . .	106
Memory Management . . . . .	106
Visibility of Data Changes . . . . .	107
Examples . . . . .	107
Chapter 16. Large Objects . . . . .	110
Historical Note . . . . .	110
Implementation Features . . . . .	110
Interfaces . . . . .	110
Creating a Large Object . . . . .	111
Importing a Large Object . . . . .	111
Exporting a Large Object . . . . .	111
Opening an Existing Large Object . . . . .	111
Writing Data to a Large Object . . . . .	111
Reading Data from a Large Object . . . . .	112
Seeking on a Large Object . . . . .	112
Closing a Large Object Descriptor . . . . .	112
Built in registered functions . . . . .	112
Accessing Large Objects from LIBPQ . . . . .	112
Sample Program . . . . .	113
Chapter 17. libpq - C Library . . . . .	118
Database Connection Functions . . . . .	118
Query Execution Functions . . . . .	124
Asynchronous Query Processing . . . . .	128
Fast Path . . . . .	130
Asynchronous Notification . . . . .	131
Functions Associated with the COPY Command . . . . .	132
libpq Tracing Functions . . . . .	134
libpq Control Functions . . . . .	134
Environment Variables . . . . .	135
Threading Behavior . . . . .	135
Sample Programs . . . . .	136
Sample Program 1 . . . . .	136
Sample Program 2 . . . . .	138
Sample Program 3 . . . . .	140
Chapter 18. libpq - C++ Binding Library . . . . .	145
Control and Initialization . . . . .	145
Environment Variables . . . . .	145
libpq++ Classes . . . . .	146
Connection Class: PgConnection . . . . .	146
Database Class: PgDatabase . . . . .	146
Database Connection Functions . . . . .	146
Query Execution Functions . . . . .	147
Asynchronous Notification . . . . .	151
Functions Associated with the COPY Command . . . . .	151
Chapter 19. pgsql - TCL Binding Library . . . . .	153
Commands . . . . .	153
Examples . . . . .	154
pgsql Command Reference Information . . . . .	155
pg_connect . . . . .	155
Name . . . . .	155
Synopsis . . . . .	155
Description . . . . .	155
Usage . . . . .	156
pg_disconnect . . . . .	156
Name . . . . .	156
Synopsis . . . . .	156
Description . . . . .	156
pg_conndefaults . . . . .	157
Name . . . . .	157

Synopsis. . . . .	157
Description . . . . .	157
Usage. . . . .	157
pg_exec . . . . .	158
Name . . . . .	158
Synopsis. . . . .	158
Description . . . . .	158
pg_result . . . . .	159
Name . . . . .	159
Synopsis. . . . .	159
Description . . . . .	160
pg_select . . . . .	161
Name . . . . .	161
Synopsis. . . . .	161
Description . . . . .	161
Usage. . . . .	161
pg_listen . . . . .	162
Name . . . . .	162
Synopsis. . . . .	162
Description . . . . .	162
pg_lo_creat . . . . .	163
Name . . . . .	163
Synopsis. . . . .	163
Description . . . . .	163
Usage. . . . .	163
pg_lo_open . . . . .	164
Name . . . . .	164
Synopsis. . . . .	164
Description . . . . .	164
Usage. . . . .	164
pg_lo_close . . . . .	165
Name . . . . .	165
Synopsis. . . . .	165
Description . . . . .	165
Usage. . . . .	165
pg_lo_read . . . . .	165
Name . . . . .	165
Synopsis. . . . .	165
Description . . . . .	166
Usage. . . . .	166
pg_lo_write . . . . .	166
Name . . . . .	166
Synopsis. . . . .	166
Description . . . . .	166
Usage. . . . .	166
pg_lo_lseek . . . . .	167
Name . . . . .	167
Synopsis. . . . .	167
Description . . . . .	167
Usage. . . . .	167
pg_lo_tell . . . . .	168
Name . . . . .	168
Synopsis. . . . .	168
Description . . . . .	168
Usage. . . . .	168
pg_lo_unlink . . . . .	168
Name . . . . .	168
Synopsis. . . . .	168
Description . . . . .	169
Usage. . . . .	169

pg_lo_import	169
Name	169
Synopsis	169
Description	169
Usage	169
pg_lo_export	170
Name	170
Synopsis	170
Description	170
Usage	170
Chapter 20. libpq - Simplified C Binding Library	171
Chapter 21. ecpg - Embedded SQL in C	172
Why Embedded SQL?	172
The Concept	172
How To Use ecpg	172
Preprocessor	172
Library	173
Error handling	173
Limitations	175
Porting From Other RDBMS Packages	176
Installation	176
For the Developer	176
ToDo List	176
The Preprocessor	177
A Complete Example	181
The Library	181
Chapter 22. ODBC Interface	183
Background	183
Windows Applications	183
Writing Applications	183
Unix Installation	184
Building the Driver	184
Configuration Files	187
ApplixWare	189
Configuration	189
Common Problems	190
Debugging ApplixWare ODBC Connections	190
Running the ApplixWare Demo	191
Useful Macros	192
Supported Platforms	192
Chapter 23. JDBC Interface	193
Building the JDBC Interface	193
Compiling the Driver	193
Installing the Driver	193
Preparing the Database for JDBC	194
Using the Driver	194
Importing JDBC	194
Loading the Driver	194
Connecting to the Database	195
Issuing a Query and Processing the Result	196
Using the Statement Interface	196
Using the ResultSet Interface	196
Performing Updates	197
Closing the Connection	197
Using Large Objects	197
Postgres Extensions to the JDBC API	198
Further Reading	236
Chapter 24. Lisp Programming Interface	237
Chapter 25. Postgres Source Code	238
Formatting	238

Chapter 26. Overview of PostgreSQL Internals . . . . .	239
The Path of a Query . . . . .	239
How Connections are Established . . . . .	240
The Parser Stage . . . . .	240
Parser . . . . .	240
Transformation Process . . . . .	242
The Postgres Rule System . . . . .	242
The Rewrite System . . . . .	242
Planner/Optimizer . . . . .	244
Generating Possible Plans . . . . .	244
Data Structure of the Plan . . . . .	244
Executor . . . . .	245
Chapter 27. pg_options . . . . .	246
Chapter 28. Genetic Query Optimization in Database Systems . . . . .	249
Query Handling as a Complex Optimization Problem . . . . .	249
Genetic Algorithms (GA) . . . . .	249
Genetic Query Optimization (GEQO) in Postgres . . . . .	250
Future Implementation Tasks for Postgres GEQO . . . . .	251
Basic Improvements . . . . .	251
References . . . . .	251
Chapter 29. Frontend/Backend Protocol . . . . .	253
Overview . . . . .	253
Protocol . . . . .	253
Startup . . . . .	254
Query . . . . .	255
Function Call . . . . .	256
Notification Responses . . . . .	257
Cancelling Requests in Progress . . . . .	257
Termination . . . . .	258
Message Data Types . . . . .	258
Message Formats . . . . .	259
Chapter 30. Postgres Signals . . . . .	267
Chapter 31. gcc Default Optimizations . . . . .	269
Chapter 32. Backend Interface . . . . .	270
BKI File Format . . . . .	270
General Commands . . . . .	270
Macro Commands . . . . .	271
Debugging Commands . . . . .	271
Example . . . . .	272
Chapter 33. Page Files . . . . .	273
Page Structure . . . . .	273
Files . . . . .	274
Bugs . . . . .	274
Appendix DG1. The CVS Repository . . . . .	275
CVS Tree Organization . . . . .	275
Getting The Source Via Anonymous CVS . . . . .	276
Getting The Source Via CVSSup . . . . .	277
Preparing A CVSSup Client System . . . . .	278
Running a CVSSup Client . . . . .	278
Installing CVSSup . . . . .	280
Installation from Sources . . . . .	281
Appendix DG2. Documentation . . . . .	283
Documentation Roadmap . . . . .	283
The Documentation Project . . . . .	284
Documentation Sources . . . . .	284
Document Structure . . . . .	284
Styles and Conventions . . . . .	285
SGML Authoring Tools . . . . .	285
Building Documentation . . . . .	287
Manpages . . . . .	287

Hardcopy Generation for v7.0 . . . . .	288
Text Hardcopy . . . . .	288
Postscript Hardcopy . . . . .	289
Toolsets . . . . .	290
Linux RPM Installation . . . . .	291
FreeBSD Installation . . . . .	291
Debian Installation . . . . .	292
Manual Installation of Tools . . . . .	292
Alternate Toolsets . . . . .	297
Bibliography . . . . .	298
SQL Reference Books . . . . .	298
PostgreSQL-Specific Documentation . . . . .	298
Proceedings and Articles . . . . .	299

# List of Tables

3-1. Postgres System Catalogs .....	12
4-1. Equivalent C Types for Built-In Postgres Types.....	20
9-1. Index Schema.....	64
9-2. B-tree Strategies.....	65
19-1. <code>pgtcl</code> Commands.....	153
30-1. Postgres Signals .....	267
33-1. Sample Page Layout.....	273
DG2-1. Postgres Documentation Products .....	283

# List of Figures

2-1. How a connection is established .....	9
3-1. The major Postgres system catalogs.....	13

# List of Examples

26-1. A Simple Select.....	241
----------------------------	-----

# Summary

Postgres, developed originally in the UC Berkeley Computer Science Department, pioneered many of the object-relational concepts now becoming available in some commercial databases. It provides SQL92/SQL3 language support, transaction integrity, and type extensibility. PostgreSQL is an open-source descendant of this original Berkeley code.

# Chapter 1. Introduction

This document is the programmer's manual for the PostgreSQL (<http://postgresql.org/>) database management system, originally developed at the University of California at Berkeley. PostgreSQL is based on Postgres release 4.2 (<http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html>). The Postgres project, led by Professor Michael Stonebraker, has been sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc.

The first part of this manual explains the Postgres approach to extensibility and describe how users can extend Postgres by adding user-defined types, operators, aggregates, and both query language and programming language functions. After a discussion of the Postgres rule system, we discuss the trigger and SPI interfaces. The manual concludes with a detailed description of the programming interfaces and support libraries for various languages.

We assume proficiency with Unix and C programming.

## Resources

This manual set is organized into several parts:

### Tutorial

An introduction for new users. Does not cover advanced features.

### User's Guide

General information for users, including available commands and data types.

### 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 management information. List of supported machines.

### Developer's Guide

Information for Postgres developers. This is intended for those who are contributing to the Postgres project; application development information should appear in the *Programmer's Guide*. Currently included in the *Programmer's Guide*.

### Reference Manual

Detailed reference information on command syntax. Currently included in the *User's Guide*.

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

### man pages

The man pages have general information on command syntax.

## FAQs

The Frequently Asked Questions (FAQ) documents address both general issues and some platform-specific issues.

## READMEs

README files are available for some contributed packages.

## Web Site

The Postgres ([postgres.org](http://postgres.org)) web site might have some information not appearing in the distribution. There is a mhonarc catalog of mailing list traffic which is a rich resource for many topics.

## Mailing Lists

The `pgsql-general` (<mailto:pgsql-general@postgresql.org>) (archive (<http://www.PostgreSQL.ORG/mhonarc/pgsql-general/>)) mailing list is a good place to have user questions answered. Other mailing lists are available; consult the Info Central section of the PostgreSQL web site for details.

## Yourself!

Postgres is an open source product. As such, it depends on the user community for ongoing support. As you begin to use Postgres, 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 it.

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` (<mailto:pgsql-docs@postgresql.org>) (archive (<http://www.PostgreSQL.ORG/mhonarc/pgsql-docs/>)) mailing list is the place to get going.

# Terminology

In the following documentation, *site* may be interpreted as the host machine on which Postgres is installed. Since it is possible to install more than one set of Postgres databases on a single host, this term more precisely denotes any particular set of installed Postgres binaries and databases.

The Postgres *superuser* is the user named `postgres` who owns the Postgres binaries and database files. As the database superuser, all protection mechanisms may be bypassed and any data accessed arbitrarily. In addition, the Postgres superuser is allowed to execute some support programs which are generally not available to all users. Note that the Postgres superuser is *not* the same as the Unix superuser (which will be referred to as *root*). The superuser should have a non-zero user identifier (*UID*) for security reasons.

The *database administrator* or DBA, is the person who is responsible for installing Postgres with mechanisms to enforce a security policy for a site. The DBA can add new users by the method described below and maintain a set of template databases for use by `createdb`.

The `postmaster` is the process that acts as a clearing-house for requests to the Postgres system. Frontend applications connect to the `postmaster`, which keeps tracks of any system

errors and communication between the backend processes. The postmaster can take several command-line arguments to tune its behavior. However, supplying arguments is necessary only if you intend to run multiple sites or a non-default site.

The Postgres backend (the actual executable program `postgres`) may be executed directly from the user shell by the Postgres super-user (with the database name as an argument). However, doing this bypasses the shared buffer pool and lock table associated with a postmaster/site, therefore this is not recommended in a multiuser site.

## Notation

... `or/usr/local/pgsql/` at the front of a file name is used to represent the path to the Postgres superuser's home directory.

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.

In examples, parentheses ( `(` and `)` ) are used to group boolean expressions. `|` is the boolean operator OR.

Examples will show commands executed from various accounts and programs. Commands executed from the root account will be preceded with `>`. Commands executed from the Postgres superuser account will be preceded with `%`, while commands executed from an unprivileged user's account will be preceded with `$`. SQL commands will be preceded with `=>` or will have no leading prompt, depending on the context.

**Note:** At the time of writing (Postgres v7.0) the notation for flagging commands is not universally consistent throughout the documentation set. Please report problems to the Documentation Mailing List (<mailto:docs@postgresql.org>).

## Problem Reporting Guidelines

When you encounter a problem in PostgreSQL we want to hear about it. Your bug reports are 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's simply too hard and there are more important things on the agenda. If you need help immediately, consider obtaining a commercial support contract.

## Identifying Bugs

Before you ask "Is this 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's a bug in the documentation. If it turns out that the program does something different from what the

documentation says, that's 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 that must be fixed outside of Postgres).

A program produces the wrong output for any given input.

A program refuses to accept valid input.

A program accepts invalid input without a notice or error message.

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 SQL is not a bug 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 can't decode the information on the TODO list, report your problem. The least we can do is make the TODO list clearer.

## 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 doesn't matter or the report would ring a bell anyway.

The following items should be contained in every bug report:

The exact sequence of steps *from program startup* 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 decode 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` startup file.) You are encouraged to minimize the size of your example, but this is not absolutely necessary. If the bug is reproduceable, we'll find it either way.

If your application uses some other client interface, such as PHP, then please try to isolate the offending queries. We probably won't 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.

The output you got. Please do not say that it didn't work or failed. If there is an error message, show it, even if you don't 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 provided by the client might not contain all the information available. In that case, also look at the output of the database server. If you do not keep your server 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 okay and is exactly what we expected. We shouldn't 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 startup options, including concerned environment variables or configuration files that you changed from the default. Again, be exact. If you are using a pre-packaged 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 what version you are currently running. If this function does not exist, say so, then we know that your version is old enough. If you can't start up the server or a client, look into the README file in the source directory or at the name of your distribution file or package name. If your version is older than 7.0 we will almost certainly tell you to upgrade. There are tons of bug fixes in each new version, that's why we write them.

If you run a pre-packaged 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.

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 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's 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 can't 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'll find that out soon enough.

When writing a bug report, please choose non-confusing terminology. The software package as such is called `PostgreSQL`, sometimes `Postgres` for short. (Sometimes the abbreviation `Psql` is used but don't do that.) When you are specifically talking about the backend server, mention that, don't just say `Postgres crashes`. The interactive frontend is called `psql` and is for all intents and purposes completely separate from the backend.

## Where to report bugs

In general, send bug reports to `pgsql-bugs@postgresql.org` (`mailto:pgsql-bugs@postgresql.org`). You are invited to find a descriptive subject for your email message, perhaps parts of the error message.

Do not send bug reports to any of the user mailing lists, such as `pgsql-sql@postgresql.org` (`mailto:pgsql-sql@postgresql.org`) or `pgsql-general@postgresql.org` (`mailto: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 `pgsql-hackers@postgresql.org` (`mailto: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 it, if the bug needs more review.

If you have a problem with the documentation, send email to `pgsql-docs@postgresql.org` (`mailto:pgsql-docs@postgresql.org`). Mention the document, chapter, and sections in your problem report.

If your bug is a portability problem on a non-supported platform, send mail to `pgsql-ports@postgresql.org` (`mailto: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 them in order to be allowed to post. If you simply want to send mail but do not want to receive list traffic, you can subscribe to the special `pgsql-loophole` list, which allows you to post to all PostgreSQL mailing lists without receiving any messages. Send email to `pgsql-loophole-request@postgresql.org` (`mailto:pgsql-loophole-request@postgresql.org`) to subscribe.

## Y2K Statement

**Author:** Written by Thomas Lockhart (`mailto:lockhart@alumni.caltech.edu`) on 1998-10-22. Updated 2000-03-31.

The PostgreSQL Global Development Team provides the Postgres 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 Postgres support team since November, 1996, is not aware of any problems in the Postgres 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 Postgres. 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 Postgres makes about dates specified with a two-digit year are documented in the current User's Guide (<http://www.postgresql.org/docs/user/datatype.htm>) 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 Postgres.

Refer to The Gnu Project (<http://www.gnu.org/software/year2000.html>) and The Perl Institute (<http://language.perl.com/news/y2k.html>) for further discussion of Y2K issues, particularly as it relates to open source, no fee software.

## Copyrights and Trademarks

PostgreSQL is Copyright © 1996-2000 by PostgreSQL Inc. and is distributed under the terms of the Berkeley license.

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.

All trademarks are the property of their respective owners.

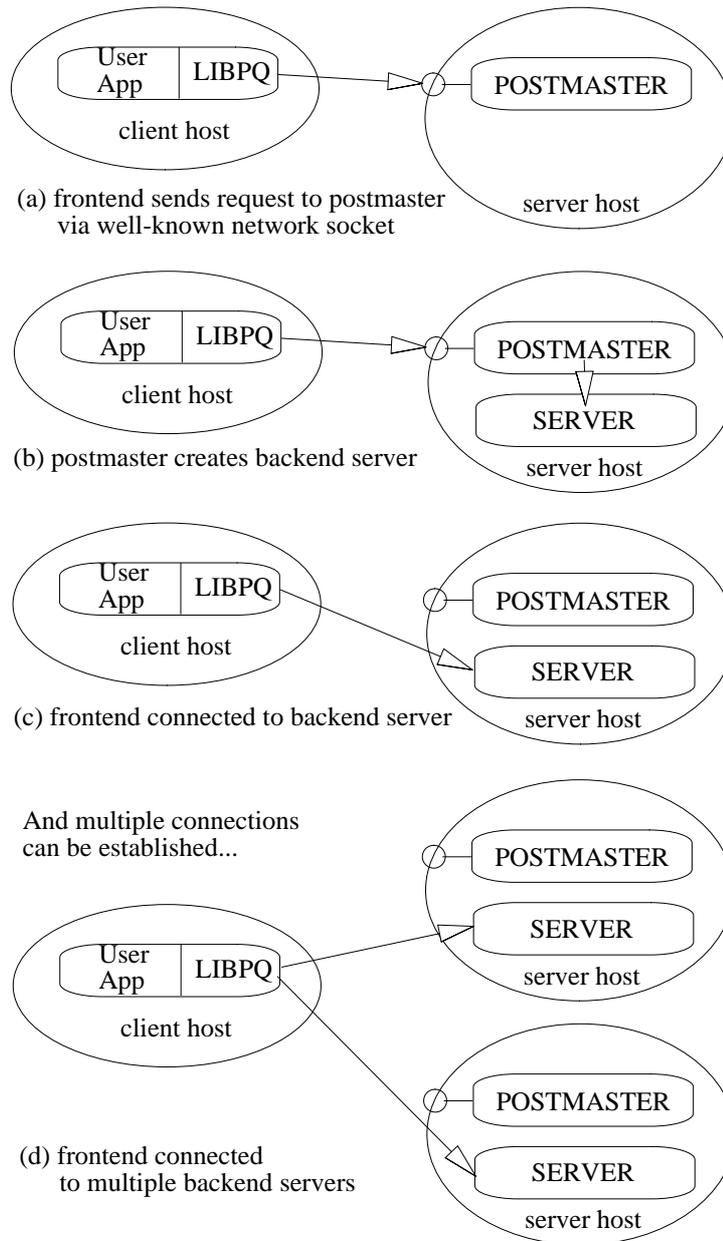
# Chapter 2. Architecture

## Postgres Architectural Concepts

Before we continue, you should understand the basic Postgres system architecture. Understanding how the parts of Postgres interact will make the next chapter somewhat clearer. In database jargon, Postgres uses a simple "process per-user" client/server model. A Postgres session consists of the following cooperating Unix processes (programs):

- A supervisory daemon process (postmaster),
- the user's frontend application (e.g., the psql program), and
- the 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 an installation or site. Frontend applications that wish to access a given database within an installation make calls to the library. The library sends user requests over the network to the postmaster (*How a connection is established(a)*), which in turn starts a new backend server process (*How a connection is established(b)*)

**Figure 2-1. How a connection is established**

and connects the frontend process to the new server (*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 requests, whereas frontend and backend processes come and go. The `libpq` library allows a single frontend to make multiple connections to backend processes. However, the frontend application is still a single-threaded process. Multithreaded frontend/backend connections are not currently supported in `libpq`. 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 filename) on the database server machine. You should also be aware that the postmaster and postgres servers run with the user-id of the Postgres

"superuser." Note that the Postgres superuser does not have to be a special user (e.g., a user named "postgres"), although many systems are installed that way. Furthermore, the Postgres superuser should definitely not be the Unix superuser, "root"! In any case, all files relating to a database should belong to this Postgres superuser.

# Chapter 3. Extending SQL: An Overview

In the sections that follow, we will discuss how you can extend the Postgres SQL query language by adding:

- functions
- types
- operators
- aggregates

## How Extensibility Works

Postgres 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 classes, like any other, but the DBMS stores its internal bookkeeping in them. One key difference between Postgres and standard relational systems is that Postgres 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 classes can be modified by the user, and since Postgres bases its internal operation on these classes, this means that Postgres 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.

Postgres 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 compiled .o file or shared library) that implements a new type or function and Postgres will load it as required. Code written in SQL are even more trivial to add to the server. This ability to modify its operation "on the fly" makes Postgres uniquely suited for rapid prototyping of new applications and storage structures.

## The Postgres Type System

The Postgres 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"; Postgres 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 class. EMP is an example of a composite type.

Postgres stores these types in only one way (within the file that stores all instances of the class) but the user can "look inside" at the attributes of these types from the query language and optimize their retrieval by (for example) defining indices on the attributes. Postgres 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 below.

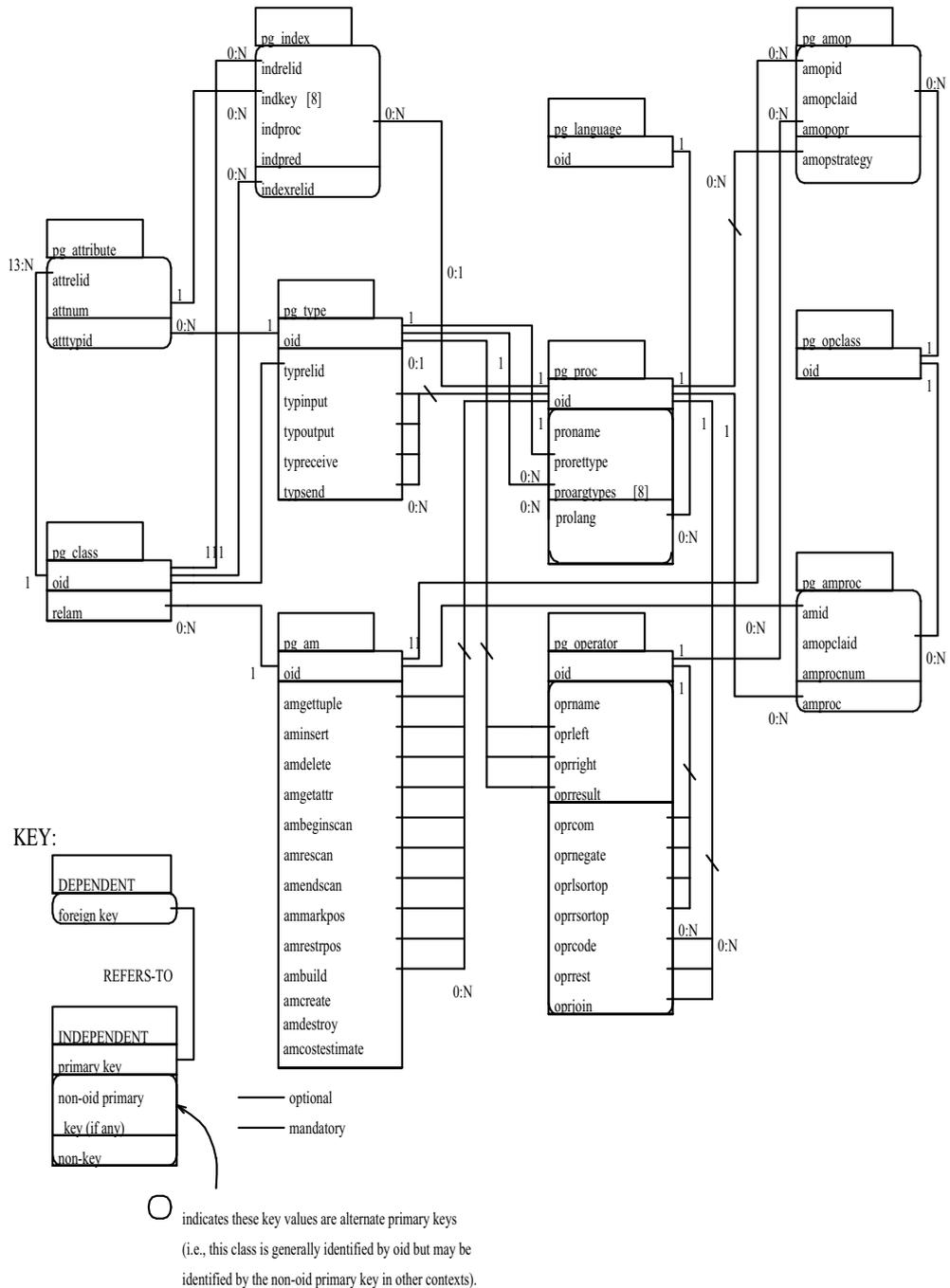
## About the Postgres System Catalogs

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 classes 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 3-1. Postgres System Catalogs**

<b>Catalog Name</b>	<b>Description</b>
pg_database	databases
pg_class	classes
pg_attribute	class attributes
pg_index	secondary indices
pg_proc	procedures (both C and SQL)
pg_type	types (both base and complex)
pg_operator	operators
pg_aggregate	aggregates and aggregate functions
pg_am	access methods
pg_amop	access method operators
pg_amproc	access method support functions
pg_opclass	access method operator classes

Figure 3-1. The major Postgres system catalogs



The Reference Manual gives a more detailed explanation of these catalogs and their attributes. However, *The major Postgres system catalogs* shows the major entities and their relationships in the system catalogs. (Attributes 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 attributes used in the queries form foreign keys in other classes.

Many different features (classes, attributes, 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 instances in one or both of these classes. For example, Postgres frequently uses type signatures (e.g., of functions and operators) to identify unique instances of other catalogs.

There are many attributes 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 the section on interfacing types and operators to indices) after we have discussed basic extensions.

# Chapter 4. Extending SQL: Functions

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 Postgres before describing how to add new types.

Postgres SQL provides three types of functions:

- query language functions (functions written in SQL)
- procedural language functions (functions written in, for example, PLTCL or PLSQL)
- programming language functions (functions written in a compiled programming language such as C)

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`.

## Query Language (SQL) Functions

SQL functions execute an arbitrary list of SQL queries, returning the results of the last query in the list. SQL functions in general return sets. If their returntype is not specified as a `setof`, then an arbitrary element of the last query's result will be returned.

The body of a SQL function following `AS` should be a list of queries separated by whitespace characters and bracketed within quotation marks. Note that quotation marks used in the queries must be escaped, by preceding them with two backslashes.

Arguments to the SQL function may be referenced in the queries using a `$n` syntax: `$1` refers to the first argument, `$2` to the second, and so on. If an argument is complex, then a *dot* notation (e.g. "`$1.emp`") may be used to access attributes of the argument or to invoke functions.

## Examples

To illustrate a simple SQL function, consider the following, which might be used to debit a bank account:

```
create function TP1 (int4, float8) returns int4
as 'update BANK set balance = BANK.balance - $2
   where BANK.acctountno = $1
   select(x = 1)'
language 'sql';
```

A user could execute this function to debit account 17 by \$100.00 as follows:

```
select (x = TP1( 17,100.0));
```

The following more interesting example takes a single argument of type EMP, and retrieves multiple results:

```
select function hobbies (EMP) returns set of HOBBIES
as 'select (HOBBIES.all) from HOBBIES
   where $1.name = HOBBIES.person'
language 'sql';
```

## SQL Functions on Base Types

The simplest possible SQL function has no arguments and simply returns a base type, such as int4:

```
CREATE FUNCTION one() RETURNS int4
AS 'SELECT 1 as RESULT' LANGUAGE 'sql';

SELECT one() AS answer;
```

```
+-----+
|answer |
+-----+
| 1     |
+-----+
```

Notice that we defined a target list for the function (with the name RESULT), but the target list of the query that invoked the function overrode the function's target list. Hence, the result is labelled answer instead of one.

It's 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(int4, int4) RETURNS int4
AS 'SELECT $1 + $2;' LANGUAGE 'sql';

SELECT add_em(1, 2) AS answer;
```

```
+-----+
|answer |
+-----+
| 3     |
+-----+
```

## SQL Functions on Composite Types

When specifying functions with arguments of composite types (such as EMP), 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, take the function `double_salary` that computes what your salary would be if it were doubled:

```
CREATE FUNCTION double_salary(EMP) RETURNS int4
AS 'SELECT $1.salary * 2 AS salary;' LANGUAGE 'sql';

SELECT name, double_salary(EMP) AS dream
FROM EMP
WHERE EMP.cubicle ~= '(2,1)::point;
```

```
+-----+-----+
|name | dream |
+-----+-----+
|Sam  | 2400 |
+-----+-----+
```

Notice the use of the syntax `$1.salary`. Before launching into the subject of functions that return composite types, we must first introduce the function notation for projecting attributes. The simple way to explain this is that we can usually use the notation `attribute(class)` and `class.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       |
+-----+
```

As we shall see, however, this is not always the case. This function notation is important when we want to use a function that returns a single instance. We do this by assembling the entire instance within the function, attribute by attribute. This is an example of a function that returns a single EMP instance:

```
CREATE FUNCTION new_emp() RETURNS EMP
AS 'SELECT \'None\'::text AS name,
    1000 AS salary,
    25 AS age,
    \'(2,2)\ '::point 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. Defining a function like this can be tricky. Some of the more important caveats are as follows:

The target list order must be exactly the same as that in which the attributes appear in the CREATE TABLE statement (or when you execute a .\* query).

You must typecast the expressions (using ::) very carefully or you will see the following error:

```
WARN::function declared to return type EMP does not retrieve (EMP.*)
```

When calling a function that returns an instance, we cannot retrieve the entire instance. We must either project an attribute out of the instance or pass the entire instance into another function.

```
SELECT name(new_emp()) AS nobody;
```

```
+-----+
|nobody |
+-----+
|None   |
+-----+
```

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 other (dot) syntax for projection when combined with function calls.

```
SELECT new_emp().name AS nobody;
WARN:parser: syntax error at or near "."
```

Any collection of commands in the SQL query language can be packaged together and defined as a function. The commands can include updates (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 returntype.

```
CREATE FUNCTION clean_EMP () RETURNS int4
AS 'DELETE FROM EMP WHERE EMP.salary <= 0;
SELECT 1 AS ignore_this'
LANGUAGE 'sql';

SELECT clean_EMP();
```

```
+---+
|x |
+---+
|1 |
+---+
```

## Procedural Language Functions

Procedural languages aren't built into Postgres. They are offered by loadable modules. Please refer to the documentation for the PL in question for details about the syntax and how the AS clause is interpreted by the PL handler.

There are two procedural languages available with the standard Postgres distribution (PLTCL and PLSQL), and other languages can be defined. Refer to *Procedural Languages* for more information.

## Internal Functions

Internal functions are functions written in C which have been statically linked into the Postgres backend process. The AS clause gives 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 AS string 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 as SQL functions during database initialization, but a user could use **CREATE FUNCTION** to create additional alias names for an internal function.

## Compiled (C) Language Functions

Functions written in C can be compiled into dynamically loadable objects, and used to implement user-defined SQL functions. The first time the user defined function is called inside the backend, the dynamic loader loads the function's object code into memory, and links the function with the running Postgres executable. The SQL syntax for **CREATE FUNCTION** links the SQL function to the C source function in one of two ways. If the SQL function has the same name as the C source function the first form of the statement is used. The string argument in the AS clause is the full pathname of the file that contains the dynamically loadable compiled object. If the name of the C function is different from the desired name of the SQL function, then the second form is used. In this form the AS clause takes two string arguments, the first is the full pathname of the dynamically loadable object file, and the second is the link symbol that the dynamic loader should search for. This link symbol is just the function name in the C source code.

**Note:** After it is used for the first time, a dynamically loaded, user function is retained in memory, and future calls to the function only incur the small overhead of a symbol table lookup.

The string which specifies the object file (the string in the AS clause) should be the *full path* of the object code file for the function, bracketed by quotation marks. If a link symbol is used in the AS clause, the link symbol should also be bracketed by single quotation marks, and should be exactly the same as the name of the function in the C source code. On Unix systems the command **nm** will print all of the link symbols in a dynamically loadable object. (Postgres will not compile a function automatically; it must be compiled before it is used in a **CREATE FUNCTION** command. See below for additional information.)

## C Language Functions on Base Types

The following table gives the C type required for parameters in the C functions that will be loaded into Postgres. The "Defined In" column gives the actual header file (in the `.../src/backend/` directory) that the equivalent C type is defined. However, if you include `utils/builtins.h`, these files will automatically be included.

**Table 4-1. Equivalent C Types for Built-In Postgres Types**

Built-In Type	C Type	Defined In
abstime	AbsoluteTime	utils/nabstime.h
bool	bool	include/c.h
box	(BOX *)	utils/geo-decls.h
bytea	(bytea *)	include/postgres.h
char	char	N/A
cid	CID	include/postgres.h
datetime	(DateTime *)	include/c.h or include/postgres.h
int2	int2	include/postgres.h
int2vector	(int2vector *)	include/postgres.h
int4	int4	include/postgres.h
float4	float32 or (float4 *)	include/c.h or include/postgres.h
float8	float64 or (float8 *)	include/c.h or include/postgres.h
lseg	(LSEG *)	include/geo-decls.h
name	(Name)	include/postgres.h
oid	oid	include/postgres.h
oidvector	(oidvector *)	include/postgres.h
path	(PATH *)	utils/geo-decls.h
point	(POINT *)	utils/geo-decls.h
regproc	regproc or REGPROC	include/postgres.h
reltime	RelativeTime	utils/nabstime.h
text	(text *)	include/postgres.h
tid	ItemPointer	storage/itemptr.h
timespan	(TimeSpan *)	include/c.h or include/postgres.h
tinterval	TimeInterval	utils/nabstime.h

Built-In Type	C Type	Defined In
uint2	uint16	include/c.h
uint4	uint32	include/c.h
xid	(XID *)	include/postgres.h

Internally, Postgres 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 Postgres can operate on it. That is, Postgres 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 (even if your computer supports by-value types of other sizes). Postgres itself only passes integer types by value. 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 (though not on most personal computers). A reasonable implementation of the `int4` type on Unix machines might be:

```
/* 4-byte integer, passed by value */
typedef int int4;
```

On the other hand, fixed-length types of any size may be passed by-reference. For example, here is a sample implementation of a Postgres 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 Postgres functions. 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 is not long enough to hold all possible strings; it's impossible to declare such a structure in C. When manipulating variable-length types, we must be careful to allocate the correct amount of memory and initialize the length field. 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;
memmove(destination->data, buffer, 40);
...
```

Now that we've gone over all of the possible structures for base types, we can show some examples of real functions. Suppose `funcs.c` look like:

```
#include <string.h>
#include "postgres.h"

/* By Value */

int
add_one(int arg)
{
    return(arg + 1);
}

/* By Reference, Fixed Length */

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));
        memset(new_t, 0, VARSIZE(t));
        VARSIZE(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);

    memset((void *) new_text, 0, new_text_size);
    VARSIZE(new_text) = new_text_size;
    strncpy(VARDATA(new_text), VARDATA(arg1),
VARSIZE(arg1)-VARHDRSZ);
    strncat(VARDATA(new_text), VARDATA(arg2),
VARSIZE(arg2)-VARHDRSZ);
    return (new_text);
}

```

On OSF/1 we would type:

```

CREATE FUNCTION add_one(int4) RETURNS int4
AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';

CREATE FUNCTION makepoint(point, point) RETURNS point
AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';

CREATE FUNCTION concat_text(text, text) RETURNS text
AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';

CREATE FUNCTION copytext(text) RETURNS text
AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';

```

On other systems, we might have to make the filename end in .sl (to indicate that it's a shared library).

## C Language Functions on Composite Types

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, Postgres provides a procedural interface for accessing fields of composite types from C. As Postgres processes a set of instances, each instance 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 instance of
EMP */
           int4 limit)
{
    bool isnull = false;
    int4 salary;
    salary = (int4) GetAttributeByName(t, "salary",
&isnull);
    if (isnull)
        return (false);
    return(salary > limit);
}
```

`GetAttributeByName` is the Postgres system function that returns attributes out of the current instance. It has three arguments: the argument of type `TUPLE` passed into the function, the name of the desired attribute, and a return parameter that describes whether the attribute is null. `GetAttributeByName` will align data properly so you can cast its return value to the desired type. For example, if you have an attribute name which is of the type name, the `GetAttributeByName` call would look like:

```
char *str;
...
str = (char *) GetAttributeByName(t, "name", &isnull)
```

The following query lets Postgres know about the `c_overpaid` function:

```
* CREATE FUNCTION c_overpaid(EMP, int4) RETURNS bool
   AS 'PGROOT/tutorial/obj/funcs.so' LANGUAGE 'c';
```

While there are ways to construct new instances or modify existing instances from within a C function, these are far too complex to discuss in this manual.

## 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 Postgres. While it may be possible to load functions written in languages other than C into Postgres, 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.

C functions with base type arguments can be written in a straightforward fashion. The C equivalents of built-in Postgres types are accessible in a C file if `PGROOT/src/backend/utils/builtins.h` is included as a header file. This can be achieved by having

```
#include <utils/builtins.h>
```

at the top of the C source file.

The basic rules for building C functions are as follows:

Most of the header (include) files for Postgres should already be installed in `PGROOT/include` (see Figure 2). You should always include

```
-I$PGROOT/include
```

on your `cc` command lines. Sometimes, you may find that you require header files that are in the server source itself (i.e., you need a file we neglected to install in `include`). In those cases you may need to add one or more of

```
-I$PGROOT/src/backend
-I$PGROOT/src/backend/include
-I$PGROOT/src/backend/port/<PORTNAME>
-I$PGROOT/src/backend/obj
```

(where `<PORTNAME>` is the name of the port, e.g., `alpha` or `sparc`).

When allocating memory, use the Postgres 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 Postgres types are declared in `postgres.h`, so it's a good idea to always include that file as well. Including `postgres.h` will also include `elog.h` and `palloc.h` for you.

Compiling and loading your object code so that it can be dynamically loaded into Postgres always requires special flags. See *Linking Dynamically-Loaded Functions* for a detailed explanation of how to do it for your particular operating system.

## Function Overloading

More than one function may be defined with the same name, as long as the arguments they take are different. In other words, function names can be *overloaded*. 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.

## Name Space Conflicts

As of Postgres v7.0, 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. This is now the preferred technique to accomplish function overloading.

### Pre-v7.0

For functions written in C, the SQL name declared in **CREATE FUNCTION** must be exactly the same as the actual name of the function in the C code (hence it must be a legal C function name).

There is a subtle implication of this restriction: while the dynamic loading routines in most operating systems are more than happy to allow you to load any number of shared libraries that contain conflicting (identically-named) function names, they may in fact botch the load in interesting ways. For example, if you define a dynamically-loaded function that happens to have the same name as a function built into Postgres, the DEC OSF/1 dynamic loader causes Postgres to call the function within itself rather than allowing Postgres to call your function. Hence, if you want your function to be used on different architectures, we recommend that you do not overload C function names.

There is a clever trick to get around the problem just described. Since there is no problem overloading SQL functions, you can define 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.

Another solution is not to use dynamic loading, but to link your functions into the backend statically and declare them as **INTERNAL** functions. Then, the functions must all have distinct C names but they can be declared with the same SQL names (as long as their argument types differ, of course). This way avoids the overhead of an SQL wrapper function, at the cost of more effort to prepare a custom backend executable. (This option is only available in version 6.5 and later, since prior versions required internal functions to have the same name in SQL as in the C code.)

# Chapter 5. Extending SQL: Types

As previously mentioned, there are two kinds of types in Postgres: base types (defined in a programming language) and composite types (instances). Examples in this section up to interfacing indices can be found in `complex.sql` and `complex.c`. Composite examples are in `funcs.sql`.

## User-Defined Types

### Functions Needed for a User-Defined Type

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-delimited 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 delimited character string. Suppose we want to define a complex type which represents complex numbers. Naturally, we 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. These 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!

```
Complex *
complex_in(char *str)
{
    double x, y;
    Complex *result;
    if (sscanf(str, " ( %lf , %lf )", &x, &y) !=
2) {
        elog(WARN, "complex_in: error in parsing
        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 *) malloc(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/obj/complex.so'
LANGUAGE 'c';

CREATE FUNCTION complex_out(opaque)
RETURNS opaque
AS 'PGROOT/tutorial/obj/complex.so'
LANGUAGE 'c';

CREATE TYPE complex (
    internallength = 16,
    input = complex_in,
    output = complex_out
);
```

As discussed earlier, Postgres fully supports arrays of base types. Additionally, Postgres supports arrays of user-defined types as well. When you define a type, Postgres 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.

## Large Objects

The types discussed to this point are all "small" objects -- that is, they are smaller than 8KB in size.

**Note:** 1024 longwords == 8192 bytes. In fact, the type must be considerably smaller than 8192 bytes, since the Postgres tuple and page overhead must also fit into this 8KB limitation. The actual value that fits depends on the machine architecture.

If you require a larger type for something like a document retrieval system or for storing bitmaps, you will need to use the Postgres large object interface.

## Chapter 6. Extending SQL: Operators

Postgres 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 arguments. 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 typecast 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.

Here is an example of creating an operator for adding two complex numbers. We assume we've already created the definition of type complex. First we need a function that does the work; then we can define the operator:

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS '$PWD/obj/complex.so'
    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.

## Operator Optimization Information

**Author:** Written by Tom Lane.

A Postgres 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 Postgres. The ones described here are all the ones that release 6.5 understands.

### 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 datatype are usually each others' commutators, and operator '`+`' is usually commutative with itself. But operator '`-`' is usually not commutative with anything.

The left argument type of a commuted operator is the same as the right argument type of its commutator, and vice versa. So the name of the commutator operator is all that Postgres needs to be given to look up the commutator, and that's all that need 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 Postgres 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 Postgres 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's `pg_operator` table. This dummy entry will have valid data only for the operator name, left and right argument types, and result type, since that's all that Postgres 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 Postgres versions before 6.5, but it is now the recommended way to do things.)

## 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  $\text{NOT} (x \ B \ y)$  for all possible inputs  $x,y$ . Notice that B is also the negator of A. For example, ' $<$ ' and ' $>=$ ' are a negator pair for most datatypes. An operator can never be validly be its own negator.

Unlike COMMUTATOR, a pair of unary operators could validly be marked as each others' negators; that would mean  $(A \ x)$  equals  $\text{NOT} (B \ x)$  for all  $x$ , or the equivalent for right-unary operators.

An operator's negator must have the same left and/or right argument types as the operator itself, so just as with COMMUTATOR, only the operator name need be given in the NEGATOR clause.

Providing NEGATOR is very helpful to the query optimizer since it allows expressions like  $\text{NOT} (x = y)$  to be simplified into  $x \ \langle \rangle \ y$ . 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.

## 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

```
field 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 VACUUM 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 `scalartsel` and `scalargtsel` for comparisons on datatypes that have some sensible means of being converted into numeric scalars for range comparisons. If possible, add the datatype to those understood by the routine `convert_to_scalar()` in `src/backend/utills/adt/selfuncs.c`. (Eventually, this routine should be replaced by per-datatype functions identified through a column of the `pg_type` table; 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/utills/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.

## 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.field1 OP table2.field2
```

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:

```
eqjoinssel      for =
neqjoinssel     for <>
scalartjoinssel for < or <=
scalargtjoinssel for > or >=
areajoinssel    for 2D area-based comparisons
positionjoinssel for 2D position-based comparisons
contjoinssel    for 2D containment-based comparisons
```

## 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 datatype 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 FLOAT datatypes 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 float equality 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().

## **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 hashjoin, where the left and right data types had better be the same (or at least bitwise equivalent), it is possible to mergejoin two distinct data types so long as they are logically compatible. For example, the int2-versus-int4 equality operator is mergejoinable. We only need sorting operators that will bring both datatypes 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 datatypes equal to the current operator's left argument type, and the SORT2 operator must have both input datatypes equal to the current operator's right argument 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 mergejoinable. These restrictions are not currently checked by CREATE OPERATOR, but a merge join may fail at runtime if any are not true:

The mergejoinable 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 input datatypes as the mergejoinable 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 mergejoin will fail to work.

# Chapter 7. Extending SQL: Aggregates

Aggregate functions in Postgres 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 datatype 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.

Actually, in order to make it easier to construct useful aggregates from existing functions, an aggregate can have one or two separate state values, one or two transition functions to update those state values, and a *final function* that computes the actual aggregate result from the ending state values.

Thus there can be as many as four datatypes involved: the type of the input data items, the type of the aggregate's result, and the types of the two state values. Only the input and result datatypes are seen by a user of the aggregate.

Some state transition functions need to look at each successive input to compute the next state value, while others ignore the specific input value and simply update their internal state. (The most useful example of the second kind is a running count of the number of input items.) The Postgres aggregate machinery defines *sfunc1* for an aggregate as a function that is passed both the old state value and the current input value, while *sfunc2* is a function that is passed only the old state value.

If we define an aggregate that uses only *sfunc1*, we have an aggregate that computes a running function of the attribute values from each instance. "Sum" is an example of this kind of aggregate. "Sum" starts at zero and always adds the current instance's value to its running total. For example, if we want to make a Sum aggregate to work on a datatype for complex numbers, we only need the addition function for that datatype. The aggregate definition is:

```
CREATE AGGREGATE complex_sum (  
    sfunc1 = complex_add,  
    basetype = complex,  
    stype1 = complex,  
    initcond1 = '(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 Postgres to figure out which kind of sum to apply to a complex column.)

If we define only *sfunc2*, we are specifying an aggregate that computes a running function that is independent of the attribute values from each instance. "Count" is the most common example of this kind of aggregate. "Count" starts at zero and adds one to its running total

for each instance, ignoring the instance value. Here, we use the built-in `int4inc` routine to do the work for us. This routine increments (adds one to) its argument.

```
CREATE AGGREGATE my_count (
    sfunc2 = int4inc, -- add one
    basetype = int4,
    stype2 = int4,
    initcond2 = '0'
);

SELECT my_count(*) as emp_count from EMP;
```

```
+-----+
|emp_count|
+-----+
| 5       |
+-----+
```

"Average" is an example of an aggregate that requires both a function to compute the running sum and a function to compute the running count. When all of the instances have been processed, the final answer for the aggregate is the running sum divided by the running count. We use the `int4pl` and `int4inc` routines we used before as well as the Postgres integer division routine, `int4div`, to compute the division of the sum by the count.

```
CREATE AGGREGATE my_average (
    sfunc1 = int4pl,      -- sum
    basetype = int4,
    stype1 = int4,
    sfunc2 = int4inc,    -- count
    stype2 = int4,
    finalfunc = int4div, -- division
    initcond1 = '0',
    initcond2 = '0'
);

SELECT my_average(salary) as emp_average FROM EMP;
```

```
+-----+
|emp_average|
+-----+
| 1640      |
+-----+
```

For further details see **CREATE AGGREGATE** in *The PostgreSQL User's Guide*.

# Chapter 8. The Postgres Rule System

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 Postgres rule system can be found in [Stonebraker *et al*, *ACM*, 1990].

Some other database systems define active database rules. These are usually stored procedures and triggers and are implemented in Postgres 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 optimizer for 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 [Ong and Goh, 1990] as well as [Stonebraker *et al*, *ACM*, 1990].

## What is a Querytree?

To understand how the rule system works it is necessary to know when it is invoked and what it's input and results are.

The rule system is located between the query parser and the optimizer. It takes the output of the parser, one querytree, and the rewrite rules from the `pg_rewrite` catalog, which are querytrees too with some extra information, and creates zero or many querytrees as result. So it's 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 querytree? It is an internal representation of an SQL statement where the single parts that built it are stored separately. These querytrees are visible when starting the Postgres backend with `debuglevel 4` and typing queries into the interactive backend interface. The rule actions in the `pg_rewrite` system catalog are also stored as querytrees. They are not formatted like the debug output, but they contain exactly the same information.

Reading a querytree 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 targetlist, water and coffee powder in a rangetable and all the buttons in a qualification expression. Since SQL representations of querytrees 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.

## The Parts of a Querytree

When reading the SQL representations of the querytrees in this document it is necessary to be able to identify the parts the statement is broken into when it is in the querytree structure. The parts of a querytree are

the commandtype

This is a simple value telling which command (SELECT, INSERT, UPDATE, DELETE) produced the parsetree.

#### the rangetable

The rangetable is a list of relations that are used in the query. In a SELECT statement that are the relations given after the FROM keyword.

Every rangetable entry identifies a table or view and tells by which name it is called in the other parts of the query. In the querytree the rangetable 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 rangetables of rules have been merged in. The examples in this document will not have this situation.

#### the resultrelation

This is an index into the rangetable 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 resultrelation is the table (or view!) where the changes take effect.

#### the targetlist

The targetlist 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).

DELETE queries don't need a targetlist because they don't produce any result. In fact the optimizer will add a special entry to the empty targetlist. But this is after the rule system and will be discussed later. For the rule system the targetlist is empty.

In INSERT queries the targetlist describes the new rows that should go into the resultrelation. Missing columns of the resultrelation will be added by the optimizer with a constant NULL expression. It is the expressions in the VALUES clause or the ones from the SELECT clause on INSERT ... SELECT.

On UPDATE queries, it describes the new rows that should replace the old ones. Here now the optimizer will add missing columns by inserting expressions that put the values from the old rows into the new one. And it will add the special entry like for DELETE too. It is the expressions from the SET attribute = expression part of the query.

Every entry in the targetlist contains an expression that can be a constant value, a variable pointing to an attribute of one of the relations in the rangetable, a parameter or an expression tree made of function calls, constants, variables, operators etc.

#### the qualification

The queries qualification is an expression much like one of those contained in the targetlist 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 others

The other parts of the querytree 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. GROUP BY is a special thing when it appears in a view definition and still needs to be documented.

## Views and the Rule System

### Implementation of Views in Postgres

Views in Postgres 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 Postgres system catalogs is exactly the same as it is for a table. So for the query parsers, there is absolutely no difference between a table and a view. They are the same thing - relations. That is the important one for now.

### 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 parsetree in place instead of creating a new one. So SELECT rules are described first.

Currently, there could be only one action and it must be a 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 example 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 on 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,        -- minimum 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 AI'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 queries rangetable. The rule has no rule qualification (discussed in 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 rules action has a qualification.

The rules action is one querytree that is an exact 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 querytree) 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-> ('s14', 8, 'black', 40.0 , 'inch');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy-> ('s15', 4, 'brown', 1.0 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy-> ('s16', 0, 'brown', 0.9 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy-> ('s17', 7, 'brown', 60 , 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy-> ('s18', 1, 'brown', 40 , 'inch');
al_bundy=>
al_bundy=> SELECT * FROM shoelace;
sl_name |sl_avail|sl_color |sl_len|sl_unit |sl_len_cm
-----+-----+-----+-----+-----+-----
s11     |      5|black    |   80|cm      |      80
s12     |      6|black    |  100|cm      |     100
s17     |      7|brown    |   60|cm      |      60
s13     |      0|black    |   35|inch    |     88.9
s14     |      8|black    |   40|inch    |    101.6
s18     |      1|brown    |   40|inch    |    101.6
s15     |      4|brown    |    1|m       |     100
s16     |      0|brown    |   0.9|m       |      90
(8 rows)

```

It's the simplest SELECT AI 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 parsetree

```

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 rangetable and checks if there are rules in `pg_rewrite` for any relation. When processing the rangetable entry for `shoelace` (the only one up to now) it finds the rule `'_RETshoelace'` with the parsetree

```

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. The first step in rewriting is merging the two

rangetables. The resulting parsetree then reads

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
FROM shoelace shoelace, shoelace *OLD*,
     shoelace *NEW*, shoelace_data s,
     unit u;
```

In step 2 it adds the qualification from the rule action to the parsetree resulting in

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
FROM shoelace shoelace, shoelace *OLD*,
     shoelace *NEW*, shoelace_data s,
     unit u
WHERE bpchareq(s.sl_unit, u.un_name);
```

And in step 3 it replaces all the variables in the parsetree, that reference the rangetable entry (the one for shoelace that is currently processed) by the corresponding targetlist expressions from the rule action. This results in the final query

```
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 shoelace, shoelace *OLD*,
     shoelace *NEW*, shoelace_data s,
     unit u
WHERE bpchareq(s.sl_unit, u.un_name);
```

Turning this back into a real SQL statement a human user would type reads

```
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;
```

That was the first rule applied. While this was done, the rangetable has grown. So the rule system continues checking the range table entries. The next one is number 2 (shoelace \*OLD\*). Relation shoelace has a rule, but this rangetable entry isn't referenced in any of the variables of the parsetree, so it is ignored. Since all the remaining rangetable entries either have no rules in `pg_rewrite` or aren't referenced, it reaches the end of the rangetable. Rewriting is complete and the above is the final result given into the optimizer. The optimizer ignores the extra rangetable entries that aren't referenced by variables in the parsetree and the plan produced by the planner/optimizer would be exactly the same as if AI had typed the above SELECT query instead of the view selection.

Now we face AI 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.

AI 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 how 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|s11     |      5|      2
sh3      |      4|s17     |      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 parsetree

```

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 that one for the shoe\_ready relation and it results in the parsetree

```

SELECT rsh.shoename, rsh.sh_avail,
       rsl.sl_name, rsl.sl_avail,
       min(rsh.sh_avail, rsl.sl_avail) AS total_avail
FROM shoe_ready shoe_ready, shoe_ready *OLD*,
     shoe_ready *NEW*, shoe rsh,
     shoelace rsl
WHERE int4ge(min(rsh.sh_avail, rsl.sl_avail), 2)
     AND (bpchareq(rsl.sl_color, rsh.slcolor)
          AND float8ge(rsl.sl_len_cm, rsh.slminlen_cm)
          AND float8le(rsl.sl_len_cm, rsh.slmaxlen_cm)
         );

```

In reality the AND clauses in the qualification will be operator nodes of type AND with a left and right expression. But that makes it lesser readable as it already is, and there are more rules to apply. So I only put them into some parantheses to group them into logical units in the order they were added and we continue with the rule for relation shoe as it is the next rangetable entry that is referenced and has a rule. The result of applying it is

```

SELECT sh.shoename, sh.sh_avail, rsl.sl_name, rsl.sl_avail,
       min(sh.sh_avail, rsl.sl_avail) AS total_avail,
FROM shoe_ready shoe_ready, shoe_ready *OLD*,
     shoe_ready *NEW*, shoe rsh,
     shoelace rsl, shoe *OLD*,
     shoe *NEW*, shoe_data sh,
     unit un
WHERE (int4ge(min(sh.sh_avail, rsl.sl_avail), 2)
     AND (bpchareq(rsl.sl_color, sh.slcolor)
          AND float8ge(rsl.sl_len_cm,
                      float8mul(sh.slminlen, un.un_fact))
          AND float8le(rsl.sl_len_cm,
                      float8mul(sh.slmaxlen, un.un_fact))
         )
     )
     AND bpchareq(sh.slunit, un.un_name);

```

And finally we apply the already well known rule for shoelace (this time on a parsetree that is a little more complex) and get

```

SELECT sh.shoename, sh.sh_avail,
       s.sl_name, s.sl_avail,
       min(sh.sh_avail, s.sl_avail) AS total_avail
FROM shoe_ready shoe_ready, shoe_ready *OLD*,
     shoe_ready *NEW*, shoe_rsh,
     shoelace rsl, shoe *OLD*,
     shoe *NEW*, shoe_data sh,
     unit un, shoelace *OLD*,
     shoelace *NEW*, shoelace_data s,
     unit u
WHERE ( (int4ge(min(sh.sh_avail, s.sl_avail), 2)
        AND (bpchareq(s.sl_color, sh.slcolor)
              AND float8ge(float8mul(s.sl_len, u.un_fact),
                              float8mul(sh.slminlen,
un.un_fact))
              AND float8le(float8mul(s.sl_len, u.un_fact),
                              float8mul(sh.slmaxlen,
un.un_fact))
              )
        )
        AND bpchareq(sh.slunit, un.un_name)
        )
        AND bpchareq(s.sl_unit, u.un_name);

```

Again we reduce it to a real SQL statement that is equivalent to the final output of the rule system:

```

SELECT sh.shoename, sh.sh_avail,
       s.sl_name, s.sl_avail,
       min(sh.sh_avail, s.sl_avail) AS total_avail
FROM shoe_data sh, shoelace_data s, unit u, unit un
WHERE min(sh.sh_avail, s.sl_avail) >= 2
      AND s.sl_color = sh.slcolor
      AND s.sl_len * u.un_fact >= sh.slminlen * un.un_fact
      AND s.sl_len * u.un_fact <= sh.slmaxlen * un.un_fact
      AND sh.sl_unit = un.un_name
      AND s.sl_unit = u.un_name;

```

Recursive processing of rules rewrote one SELECT from a view into a parsetree, that is equivalent to exactly that what AI had to type if there would be no views at all.

**Note:** There is currently no recursion stopping mechanism for view rules in the rule system (only for the other 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 on the first CREATE VIEW, the second relation does not exist and thus the first view cannot select from the second.

## View Rules in Non-SELECT Statements

Two details of the parsetree aren't touched in the description of view rules above. These are the commandtype and the resultrelation. In fact, view rules don't need these informations.

There are only a few differences between a parsetree for a SELECT and one for any other command. Obviously they have another commandtype and this time the resultrelation points to the rangetable entry where the result should go. Anything else is absolutely the same. So having two tables t1 and t2 with attributes a and b, the parsetrees 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 rangetables contain entries for the tables t1 and t2.

The targetlists contain one variable that points to attribute b of the rangetable entry for table t2.

The qualification expressions compare the attributes a of both ranges for equality.

The consequence is, that both parsetrees 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 targetlist by the optimizer and the final parsetree 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 parsetree) that this is an UPDATE, and he knows that this result should go into table t1. But which of the 666 rows that are there has to be replaced by the new row? The plan executed is a join with a qualification that potentially could produce any number of rows between 0 and 666 in unknown order.

To resolve this problem, another entry is added to the targetlist in UPDATE and DELETE statements. The current tuple ID (ctid). This is a system attribute with a special feature. It contains the block and position in the block for the row. Knowing the table, the ctid can be used to find one specific row in a 1.5GB sized table containing millions of rows by fetching one single data block. After adding the ctid to the targetlist, the final result set could be defined as

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Now another detail of Postgres 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 that all, we can simply apply view rules in absolutely the same way to any command. There is no difference.

## The Power of Views in Postgres

The above demonstrates how the rule system incorporates view definitions into the original parsetree. In the second example a simple SELECT from one view created a final parsetree that is a join of 4 tables (unit is used twice with different names).

### Benefits

The benefit of implementing views with the rule system is, that the optimizer 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 parsetree. And this is still the situation when the original query is already a join over views. Now the optimizer has to decide which is the best path to execute the query. The more information the optimizer has, the better this decision can be. And the rule system as implemented in Postgres ensures, that this is all information available about the query up to now.

### Concerns

There was a long time where the Postgres rule system was considered broken. The use of rules was not recommended and the only part working was view rules. And also these view rules gave problems because the rule system wasn't able to apply them properly on statements other than a SELECT (for example an UPDATE that used data from a view didn't work).

During that time, development moved on and many features were added to the parser and optimizer. The rule system got more and more out of sync with their capabilities and it became harder and harder to start fixing it. Thus, no one did.

For 6.4, someone locked the door, took a deep breath and shuffled that damned thing up. What came out was a rule system with the capabilities described in this document. But there are still some constructs not handled and some where it fails due to things that are currently not supported by the Postgres query optimizer.

Views with aggregate columns have bad problems. Aggregate expressions in qualifications must be used in subselects. Currently it is not possible to do a join of two views, each having an aggregate column, and compare the two aggregate values in the qualification. In the meantime it is possible to put these aggregate expressions into functions with the appropriate arguments and use them in the view definition.

Views of unions are currently not supported. Well it's easy to rewrite a simple SELECT into a union. But it is a little difficult if the view is part of a join doing an update.

ORDER BY clauses in view definitions aren't supported.

DISTINCT isn't supported in view definitions.

There is no good reason why the optimizer should not handle parsetree constructs that the parser could never produce due to limitations in the SQL syntax. The author hopes that these items disappear in the future.

## Implementation Side Effects

Using the described rule system to implement views has a funny side effect. The following does not seem to work:

```
al_bundy=> INSERT INTO shoe (shoename, sh_avail, slcolor)
al_bundy->     VALUES ('sh5', 0, 'black');
INSERT 20128 1
al_bundy=> SELECT shoename, sh_avail, slcolor FROM shoe_data;
shoename |sh_avail|slcolor
-----+-----+-----
sh1      |      2|black
sh3      |      4|brown
sh2      |      0|black
sh4      |      3|brown
(4 rows)
```

The interesting thing is that the return code for INSERT gave us an object ID and told that 1 row has been inserted. But it doesn't appear in shoe\_data. Looking into the database directory we can see, that the database file for the view relation shoe seems now to have a data block. And that is definitely the case.

We can also issue a DELETE and if it does not have a qualification, it tells us that rows have been deleted and the next vacuum run will reset the file to zero size.

The reason for that behaviour is, that the parsetree for the INSERT does not reference the shoe relation in any variable. The targetlist contains only constant values. So there is no rule to apply and it goes down unchanged into execution and the row is inserted. And so for the DELETE.

To change this we can define rules that modify the behaviour of non-SELECT queries. This is the topic of the next section.

## Rules on INSERT, UPDATE and DELETE

### Differences to 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 parsetree in place. Instead they create zero or many new parsetrees and can throw away the original one.

## 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 commandtype of a parsetree are equal to the object and event given in the CREATE RULE command. For update rules, the rule system creates a list of parsetrees. Initially the parsetree 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 parsetrees for a one-action rule.

No qualification and not INSTEAD:

The parsetree from the rule action where the original parsetrees qualification has been added.

No qualification but INSTEAD:

The parsetree from the rule action where the original parsetrees qualification has been added.

Qualification given and not INSTEAD:

The parsetree from the rule action where the rule qualification and the original parsetrees qualification have been added.

Qualification given and INSTEAD:

The parsetree from the rule action where the rule qualification and the original parsetrees qualification have been added.

The original parsetree where the negated rule qualification has been added.

Finally, if the rule is not INSTEAD, the unchanged original parsetree is added to the list. Since only qualified INSTEAD rules already add the original parsetree, we end up with a total maximum of two parsetrees for a rule with one action.

The parsetrees generated from rule actions are thrown into the rewrite system again and maybe more rules get applied resulting in more or less parsetrees. So the parsetrees in the rule actions must have either another commandtype or another resultrelation. 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 aborts the transaction.

The parsetrees found in the actions of the `pg_rewrite` system catalog are only templates. Since they can reference the rangetable entries for NEW and OLD, some substitutions have to be made before they can be used. For any reference to NEW, the targetlist of the original query is searched for a corresponding entry. If found, that entries expression is placed into the reference. Otherwise NEW means the same as OLD. Any reference to OLD is replaced by a reference to the rangetable entry which is the resultrelation.

## 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 writes us entries every time and UPDATE is performed on `shoelace_data`.

```
CREATE TABLE shoelace_log (
    sl_name      char(10),      -- shoelace changed
    sl_avail     integer,      -- new available value
    log_who      name,         -- who did it
    log_when     datetime      -- 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,
    getpgusername(),
    'now'::text
);
```

One interesting detail is the casting of 'now' in the rules INSERT action to type text. Without that, the parser would see at CREATE RULE time, that the target type in `shoelace_log` is a datetime and tries to make a constant from it - with success. So a constant datetime value would be stored in the rule action and all log entries would have the time of the CREATE RULE statement. Not exactly what we want. The casting causes that the parser constructs a `datetime('now'::text)` from it and this will be evaluated when the rule is executed.

Now Al does

```
al_bundy=> UPDATE shoelace_data SET sl_avail = 6
al_bundy->      WHERE sl_name = 'sl7';
```

and we look at the logtable.

```
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 parsetree (this time the parts of the original parsetree 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 SELECT
    *NEW*.sl_name, *NEW*.sl_avail,
    getpgusername(), datetime('now'::text)
FROM shoelace_data *NEW*, shoelace_data *OLD*,
    shoelace_log shoelace_log;
```

Don't trust the output of the `pg_rules` system view. It specially handles the situation that there are only references to NEW and OLD in the INSERT and outputs the VALUES format of INSERT. In fact there is no difference between an INSERT ... VALUES and an INSERT ... SELECT on parsetree level. They both have rangetables, targetlists and maybe qualifications etc. The optimizer later decides, if to create an execution plan of type result, seqscan, indexscan, join or whatever for that parsetree. If there are no references to rangetable entries left in the parsetree, it becomes a result execution plan (the INSERT ... VALUES version). The rule action above can truly result in both variants.

The rule is a qualified non-`INSTEAD` rule, so the rule system has to return two parsetrees. The modified rule action and the original parsetree. In the first step the rangetable of the original query is incorporated into the rules action parsetree. This results in

```
INSERT INTO shoelace_log SELECT
    *NEW*.sl_name, *NEW*.sl_avai,
    getpgusername(), datetime('now'::text)
FROM shoelace_data shoelace_data, shoelace_data *NEW*,
    shoelace_data *OLD*, shoelace_log shoelace_log;
```

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 SELECT
    *NEW*.sl_name, *NEW*.sl_avai,
    getpgusername(), datetime('now'::text)
FROM shoelace_data shoelace_data, shoelace_data *NEW*,
    shoelace_data *OLD*, shoelace_log shoelace_log
WHERE int4ne(*NEW*.sl_avail, *OLD*.sl_avail);
```

In step 3 the original parsetrees qualification is added, restricting the resultset further to only the rows touched by the original parsetree.

```
INSERT INTO shoelace_log SELECT
    *NEW*.sl_name, *NEW*.sl_avai,
    getpgusername(), datetime('now'::text)
FROM shoelace_data shoelace_data, shoelace_data *NEW*,
    shoelace_data *OLD*, shoelace_log shoelace_log
WHERE int4ne(*NEW*.sl_avail, *OLD*.sl_avail)
AND bpchareq(shoelace_data.sl_name, 'sl7');
```

Step 4 substitutes NEW references by the targetlist entries from the original parsetree or with the matching variable references from the result relation.

```
INSERT INTO shoelace_log SELECT
    shoelace_data.sl_name, 6,
    getpgusername(), datetime('now'::text)
FROM shoelace_data shoelace_data, shoelace_data *NEW*,
    shoelace_data *OLD*, shoelace_log shoelace_log
WHERE int4ne(6, *OLD*.sl_avail)
    AND bpchareq(shoelace_data.sl_name, 'sl7');
```

Step 5 replaces OLD references into resultrelation references.

```
INSERT INTO shoelace_log SELECT
    shoelace_data.sl_name, 6,
    getpgusername(), datetime('now'::text)
FROM shoelace_data shoelace_data, shoelace_data *NEW*,
    shoelace_data *OLD*, shoelace_log shoelace_log
WHERE int4ne(6, shoelace_data.sl_avail)
    AND bpchareq(shoelace_data.sl_name, 'sl7');
```

That's it. So reduced to the max the return from the rule system is a list of two parsetrees that are the same as the statements:

```
INSERT INTO shoelace_log SELECT
    shoelace_data.sl_name, 6,
    getpgusername(), 'now'
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 an

```
UPDATE shoelace_data SET sl_color = 'green'
WHERE sl_name = 'sl7';
```

No log entry would get written because due to the fact that this time the original parsetree does not contain a targetlist entry for sl\_avail, NEW.sl\_avail will get replaced by shoelace\_data.sl\_avail resulting in the extra query

```
INSERT INTO shoelace_log SELECT
    shoelace_data.sl_name, shoelace_data.sl_avail,
    getpgusername(), 'now'
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. Since there is no difference on parsetree level between an INSERT ... SELECT, and an INSERT ... VALUES, it will also work if the

original query modifies multiple rows. So if AI 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 parsetrees qualification is different and that results in the extra parsetree

```
INSERT INTO shoelace_log SELECT
    shoelace_data.sl_name, 0,
    getpgusername(), 'now'
FROM shoelace_data
WHERE 0 != shoelace_data.sl_avail
AND shoelace_data.sl_color = 'black';
```

This parsetree will surely insert three new log entries. And that's absolutely correct.

It is important, that the original parsetree is executed last. The Postgres "traffic cop" does a command counter increment between the execution of the two parsetrees so the second one can see changes made by the first. 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.

## Cooperation with Views

A simple way to protect view relations from the mentioned possibility that someone can INSERT, UPDATE and DELETE invisible data on them is to let those parsetrees 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 AI 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 parsetrees 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 fact 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 or so 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 parsetree 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 partlist. 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 partlist and one with a special trick. The create commands for anything 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 that what's on the part list. We take a quick look at the current data,

```
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
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 it's description will be the last in this document (but not the last example :-). First there was the parsers 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 parsetree is trashed. Note that this query sill 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 parsetree

```
INSERT INTO shoelace_log SELECT
    s.sl_name,
    int4pl(s.sl_avail, shoelace_arrive.arr_quant),
    getpgusername(),
    datetime('now'::text)
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
    shoelace_ok *OLD*, shoelace_ok *NEW*,
    shoelace shoelace, shoelace *OLD*,
    shoelace *NEW*, shoelace_data shoelace_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, shoelace_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 parsetrees. So we end up with two final parsetrees that are equal to the SQL statements

```
INSERT INTO shoelace_log SELECT
    s.sl_name,
    s.sl_avail + shoelace_arrive.arr_quant,
    getpgusername(),
    'now'
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 rangetable where it could definitely be

reduced to one. The optimizer 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 rangetable 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 Postgres 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-> ('s19', 0, 'pink', 35.0, 'inch', 0.0);
al_bundy=> INSERT INTO shoelace VALUES
al_bundy-> ('s110', 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);
```

It's output is

```
al_bundy=> SELECT * FROM shoelace_obsolete;
sl_name  |sl_avail|sl_color  |sl_len|sl_unit  |sl_len_cm
-----+-----+-----+-----+-----+-----
s19      |      0|pink      |   35|inch     |   88.9
s110     |    1000|magenta   |   40|inch     |  101.6
```

For the 1000 magenta shoelaces we must debt AI before we can throw 'em away, but that's another problem. The pink entry we delete. To make it a little harder for Postgres, we don't delete it directly. Instead we create one more view

```
CREATE VIEW shoelace_candelelete AS
  SELECT * FROM shoelace_obsolete WHERE sl_avail = 0;
```

and do it this way:

```
DELETE FROM shoelace WHERE EXISTS
  (SELECT * FROM shoelace_candelelete
   WHERE sl_name = shoelace.sl_name);
```

Voila:

```
al_bundy=> SELECT * FROM shoelace;
sl_name  |sl_avail|sl_color  |sl_len|sl_unit  |sl_len_cm
-----+-----+-----+-----+-----+-----
s11      |      5|black     |   80|cm       |      80
s12      |      6|black     |  100|cm       |     100
s17      |      6|brown     |   60|cm       |      60
s14      |      8|black     |   40|inch     |    101.6
s13      |     10|black     |   35|inch     |     88.9
s18      |     21|brown     |   40|inch     |    101.6
s110     |    1000|magenta   |   40|inch     |    101.6
s15      |      4|brown     |    1|m        |     100
s16      |     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 parsetree 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.

## Rules and Permissions

Due to rewriting of queries by the Postgres 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 Postgres rule system changes the behaviour of the default access control system. Relations that are used due to rules get checked during the rewrite against the permissions of the relation owner, the rule is defined on. 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 still performed, so nobody than 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 successful. 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.

## Rules versus Triggers

Many things that can be done using triggers can also be implemented using the Postgres 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 parsetree 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 indices setup, the optimizer 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 optimizer 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 indices. And it depends on the overall size of the software table if the rule will still be faster in the seqscan 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 manufacturer = '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 Nestloop over two IndexScan's. 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.manufacturer = '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 Postgres version 6.4, the attribute specification for rule events is disabled (it will have it's 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 targetlist 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 it's actions will only be done when a specific attribute is updated is hidden in it's 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 it's decision. The rule system will know it by looking up the targetlist 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 significant slower than triggers if their actions result in large and bad qualified joins, a situation where the optimizer 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 9. Interfacing Extensions To Indices

The procedures described thus far let you define a new type, 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 *The major Postgres system catalogs*. The right half shows the catalogs that we must modify in order to tell Postgres 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.

The `pg_am` class contains one instance for every user defined access method. Support for the heap access method is built into Postgres, but every other access method is described here. The schema is

**Table 9-1. Index Schema**

Attribute	Description
<code>amname</code>	name of the access method
<code>amowner</code>	object id of the owner's instance in <code>pg_user</code>
<code>amstrategies</code>	number of strategies for this access method (see below)
<code>amsupport</code>	number of support routines for this access method (see below)
<code>amorderstrategy</code>	zero if the index offers no sort order, otherwise the strategy number of the strategy operator that describes the sort order
<code>amgettuple</code>	
<code>aminsert</code>	
...	procedure identifiers for interface routines to the access method. For example, <code>regproc</code> ids for opening, closing, and getting instances from the access method appear here.

The object ID of the instance in `pg_am` is used as a foreign key in lots of other classes. You don't need to add a new instance to this class; all you're interested in is the object ID of the access method instance you want to extend:

```
SELECT oid FROM pg_am WHERE amname = 'btree';
```

```
oid
----
403
(1 row)
```

We will use that **SELECT** in a **WHERE** clause later.

The `amstrategies` attribute exists to standardize comparisons across data types. For example, B-trees impose a strict ordering on keys, lesser to greater. Since Postgres allows the user to define operators, Postgres cannot look at the name of an operator (eg, ">" 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. Postgres 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 Postgres needs to know, for example, that the "<=" and ">" operators partition a B-tree. Postgres uses strategies to express these relationships between operators and the way they can be used to scan indices.

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 operator class. In the `pg_am` class, the `amstrategies` attribute is the number of strategies defined for this access method. For B-trees, this number is 5. These strategies correspond to

**Table 9-2. 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 procedures corresponding to the comparisons above 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 procedures yet; just understand that there must be a set of these procedures for `int2`, `int4`, `oid`, and every other data type on which a B-tree can operate.

Sometimes, strategies aren't enough information for the system to figure out how to use an index. Some access methods require other 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 user qualifications in SQL queries; they are administrative routines used by the access methods, internally.

In order to manage diverse support routines consistently across all Postgres access methods, `pg_am` includes an attribute called `amsupport`. This attribute 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.

**Note:** Strictly speaking, this routine can return a negative number (< 0), 0, 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 procedures 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` entry 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, `btree` has `amorderstrategy = 1` which is its "less than" strategy number.

The next class of interest is `pg_opclass`. This class exists only to associate an operator class name and perhaps a default type with an operator class oid. Some existing opclasses are `int2_ops`, `int4_ops`, and `oid_ops`. You need to add an instance with your opclass name (for example, `complex_abs_ops`) to `pg_opclass`. The oid of this instance will be a foreign key in other classes, notably `pg_amop`.

```
INSERT INTO pg_opclass (opcname, opcdeftype)
  SELECT 'complex_abs_ops', oid FROM pg_type WHERE typename =
'complex';
```

```
SELECT oid, opcname, opcdeftype
  FROM pg_opclass
  WHERE opcname = 'complex_abs_ops';
```

```
   oid   |   opcname   | opcdeftype
-----+-----+-----
 277975 | complex_abs_ops |      277946
(1 row)
```

Note that the oid for your `pg_opclass` instance 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 opclass the default index opclass for the `complex` datatype. If you don't, just insert zero into `opcdeftype`, rather than inserting the datatype's oid:

```
INSERT INTO pg_opclass (opcname, opcdeftype) VALUES
('complex_abs_ops', 0);
```

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 earlier in this manual. For the `complex_abs_ops` operator class on `Btrees`, the operators we require are:

```
absolute value less-than
absolute value less-than-or-equal
absolute value equal
absolute value greater-than-or-equal
```

```
absolute value greater-than
```

Suppose the code that implements the functions defined is stored in the file

```
PGROOT/src/tutorial/complex.c
```

Part of the C code looks like this: (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.)

```
#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);
}
```

We make the function known to Postgres like this:

```
CREATE FUNCTION complex_abs_eq(complex, complex)
    RETURNS bool
    AS 'PGROOT/tutorial/obj/complex.so'
    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 datatype 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 Postgres can cope with operators having the same name as long as they have different input datatypes, 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 datatype name in the C function name, so as not to conflict with functions for other datatypes.

Third, we could have made the Postgres name of the function `abs_eq`, relying on Postgres to distinguish it by input datatypes from any other Postgres function of the same name. To keep the example simple, we make the function have the same names at the C level and Postgres level.

Finally, note that these operator functions return Boolean values. The access methods rely on this fact. (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` attribute of the `pg_am` class. 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 complexes, and pick ours out:

```
SELECT o.oid AS opoid, o.oprname
INTO 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're ready to update `pg_amop` with our new operator class. The most important thing in this entire discussion is that the operators are ordered, from less than through greater than, in `pg_amop`. We add the instances we need:

```
INSERT INTO pg_amop (amopid, amopclaid, amopopr, amopstrategy)
SELECT am.oid, opcl.oid, c.opoid, 1
FROM pg_am am, pg_opclass opcl, complex_ops_tmp c
WHERE amname = 'btree' AND
      opcname = 'complex_abs_ops' AND
      c.oprname = '<';
```

Now do this for the other operators substituting for the "1" in the third 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 next step is 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` class, keyed by the access method `oid` and the operator class `oid`. First, we need to register the function in Postgres (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 int4
  AS 'PGROOT/tutorial/obj/complex.so'
  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 instance as follows:

```
INSERT INTO pg_amproc (amid, amopclaid, amproc, amprocnum)
  SELECT a.oid, b.oid, c.oid, 1
  FROM pg_am a, pg_opclass b, pg_proc c
  WHERE a.amname = 'btree' AND
        b.opcname = 'complex_abs_ops' AND
        c.proname = 'complex_abs_cmp';
```

And we're done! (Whew.) It should now be possible to create and use `btree` indexes on `complex` columns.

# Chapter 10. Index Cost Estimation Functions

**Author:** Written by Tom Lane (mailto: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 Postgres 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);
```

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 three parameters are pass-by-reference outputs:

\*indexStartupCost

Set to cost of index startup processing

\*indexTotalCost

Set to total cost of index processing

\*indexSelectivity

Set to index selectivity

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 "startup 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 startup 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.

### 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;

```

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

```

proretype = 0
pronargs = 7
proargtypes = 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 11. GiST Indices

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 the Berkely database site at <http://epoch.cs.berkeley.edu:8000/>.

**Author:** This extraction from an e-mail sent by Eugene Selkov Jr. (<mailto: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 (<ftp://s2k-ftp.cs.berkeley.edu/pub/gist/pggist/pggist.tgz>) 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 (<http://gist.cs.berkeley.edu:8000/gist/>)). 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 (HPUX) 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 than 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);
```

Roger that!! I thought it could be related to a number of Postgres 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 12. Procedural Languages

Postgres supports the definition of procedural languages. In the case of a function or trigger procedure defined in a procedural language, the database has no built-in knowledge about how to interpret the function's source text. Instead, the task is passed to a handler that knows the details of the 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 (PL) is outside the scope of this manual.

## Installing Procedural Languages

### Procedural Language Installation

A procedural language is installed in the database in three steps. (For the languages supplied with the standard distribution, the shell script `createlang` can be used instead of carrying out the details manually.)

1. The shared object for the language handler must be compiled and installed. By default the handler for PL/pgSQL is built and installed into the database library directory. If Tcl/Tk support is configured in, the handler for PL/Tcl is also built and installed in the same location.
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 datatypes 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  
    LANCMPILER 'description';
```

The optional keyword 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 backend, the TRUSTED flag should only be given for languages that don't allow access to database backends internals or the filesystem. The languages PL/pgSQL and PL/Tcl are known to be trusted.

### Example

1. The following command tells the database where to find the shared object for the PL/pgSQL language's call handler function.

```
CREATE FUNCTION plpgsql_call_handler () RETURNS OPAQUE AS  
    '/usr/local/pgsql/lib/plpgsql.so' LANGUAGE 'C';
```

## 2. The command

```
CREATE TRUSTED PROCEDURAL LANGUAGE 'plpgsql'  
    HANDLER plpgsql_call_handler  
    LANCOMPILER 'PL/pgSQL';
```

then defines that the previously declared call handler function should be invoked for functions and trigger procedures where the language attribute is 'plpgsql'.

PL handler functions have a special call interface that is different from regular C language functions. One of the arguments given to the handler is the object ID in the `pg_proc` tables entry for the function that should be executed. The handler examines various system catalogs to analyze the functions call arguments and its return data type. The source text of the functions body is found in the `prosrc` attribute of `pg_proc`. Due to this, PL functions can be overloaded like SQL language functions. There can be multiple different PL functions having the same function name, as long as the call arguments differ.

Procedural languages defined in the `template1` database are automatically defined in all subsequently created databases. So the database administrator can decide which languages are available by default.

# Chapter 13. Linking Dynamically-Loaded Functions

After you have created and registered a user-defined function, your work is essentially done. Postgres, however, must load the object code (e.g., a `.o` file, or a shared library) that implements your function. As previously mentioned, Postgres loads your code at runtime, as required. In order to allow your code to be dynamically loaded, you may have to compile and link-edit it in a special way. This section briefly describes how to perform the compilation and link-editing required before you can load your user-defined functions into a running Postgres server.

You should expect to read (and reread, and re-reread) the manual pages for the C compiler, `cc(1)`, and the link editor, `ld(1)`, if you have specific questions. In addition, the contrib area (`PGROOT/contrib`) and the regression test suites in the directory `PGROOT/src/test/regress` contain several working examples of this process. If you copy one of these examples then you should not have any problems.

The following terminology will be used below:

*Dynamic loading* is what Postgres does to an object file. The object file is copied into the running Postgres server and the functions and variables within the file are made available to the functions within the Postgres process. Postgres does this using the dynamic loading mechanism provided by the operating system.

*Loading and link editing* is what you do to an object file in order to produce another kind of object file (e.g., an executable program or a shared library). You perform this using the link editing program, `ld(1)`.

The following general restrictions and notes also apply to the discussion below:

Paths given to the create function command must be absolute paths (i.e., start with `"/`) that refer to directories visible on the machine on which the Postgres server is running.

**Tip:** Relative paths do in fact work, but are relative to the directory where the database resides (which is generally invisible to the frontend application). Obviously, it makes no sense to make the path relative to the directory in which the user started the frontend application, since the server could be running on a completely different machine!

The Postgres user must be able to traverse the path given to the create function command and be able to read the object file. This is because the Postgres server runs as the Postgres user, not as the user who starts up the frontend process. (Making the file or a higher-level directory unreadable and/or unexecutable by the "postgres" user is an extremely common mistake.)

Symbol names defined within object files must not conflict with each other or with symbols defined in Postgres.

The GNU C compiler usually does not provide the special options that are required to use the operating system's dynamic loader interface. In such cases, the C compiler that comes with the operating system must be used.

## Linux

Under Linux ELF, object files can be generated by specifying the compiler flag `-fpic`.

For example,

```
# simple Linux example
% cc -fpic -c foo.c
```

produces an object file called `foo.o` that can then be dynamically loaded into Postgres. No additional loading or link-editing must be performed.

## DEC OSF/1

Under DEC OSF/1, you can take any simple object file and produce a shared object file by running the `ld` command over it with the correct options. The commands to do this look like:

```
# simple DEC OSF/1 example
% cc -c foo.c
% ld -shared -expect_unresolved '*' -o foo.so foo.o
```

The resulting shared object file can then be loaded into Postgres. When specifying the object file name to the create function command, one must give it the name of the shared object file (ending in `.so`) rather than the simple object file.

**Tip:** Actually, Postgres does not care what you name the file as long as it is a shared object file. If you prefer to name your shared object files with the extension `.o`, this is fine with Postgres so long as you make sure that the correct file name is given to the create function command. In other words, you must simply be consistent. However, from a pragmatic point of view, we discourage this practice because you will undoubtedly confuse yourself with regards to which files have been made into shared object files and which have not. For example, it's very hard to write Makefiles to do the link-editing automatically if both the object file and the shared object file end in `.o`!

If the file you specify is not a shared object, the backend will hang!

## SunOS 4.x, Solaris 2.x and HP-UX

Under SunOS 4.x, Solaris 2.x and HP-UX, the simple object file must be created by compiling the source file with special compiler flags and a shared library must be produced. The necessary steps with HP-UX are as follows. The `+z` flag to the HP-UX C compiler produces so-called "Position Independent Code" (PIC) and the `+u` flag removes some alignment restrictions that the PA-RISC architecture normally enforces. The object file must be turned into a shared library using the HP-UX link editor with the `-b` option. This sounds complicated but is actually very simple, since the commands to do it are just:

```
# simple HP-UX example
```

*Functions*

```
% cc +z +u -c foo.c
% ld -b -o foo.sl foo.o
```

As with the .so files mentioned in the last subsection, the create function command must be told which file is the correct file to load (i.e., you must give it the location of the shared library, or .sl file). Under SunOS 4.x, the commands look like:

```
# simple SunOS 4.x example
% cc -PIC -c foo.c
% ld -dc -dp -Bdynamic -o foo.so foo.o
```

and the equivalent lines under Solaris 2.x are:

```
# simple Solaris 2.x example
% cc -K PIC -c foo.c
% ld -G -Bdynamic -o foo.so foo.o
```

or

```
# simple Solaris 2.x example
% gcc -fPIC -c foo.c
% ld -G -Bdynamic -o foo.so foo.o
```

When linking shared libraries, you may have to specify some additional shared libraries (typically system libraries, such as the C and math libraries) on your ld command line.

# Chapter 14. Triggers

Postgres has various client interfaces such as Perl, Tcl, Python and C, as well as three *Procedural Languages* (PL). It is also possible to call C functions as trigger actions. 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.

## Trigger Creation

If a trigger event occurs, the trigger manager (called by the Executor) initializes the global structure TriggerData \*CurrentTriggerData (described below) and calls the trigger function to handle the event.

The trigger function must be created before the trigger is created as a function taking no arguments and returns opaque.

The syntax for creating triggers is as follows:

```
CREATE TRIGGER <trigger name> <BEFORE|AFTER>
<INSERT|DELETE|UPDATE>
ON <relation name> FOR EACH <ROW|STATEMENT>
EXECUTE PROCEDURE <procedure name> (<function args>);
```

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.

The next word determines whether the function is called before or after the event.

The next element of the command determines on what event(s) will trigger the function. Multiple events can be specified separated by OR.

The relation name determines which table the event applies to.

The FOR EACH statement determines whether the trigger is fired for each affected row or before (or after) the entire statement has completed.

The procedure name is the C function called.

The args are passed to the function in the CurrentTriggerData structure. The purpose of passing arguments to the function is to allow different triggers with similar requirements to call the same function.

Also, function 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...

## Interaction with the Trigger Manager

As mentioned above, when function is called by the trigger manager, structure `TriggerData *CurrentTriggerData` is NOT NULL and initialized. So it is better to check `CurrentTriggerData` against being NULL at the start and set it to NULL just after fetching the information to prevent calls to a trigger function not from the trigger manager.

`struct TriggerData` is defined in `src/include/commands/trigger.h`:

```
typedef struct TriggerData
{
    TriggerEvent    tg_event;
    Relation        tg_relation;
    HeapTuple       tg_trigtuple;
    HeapTuple       tg_newtuple;
    Trigger         *tg_trigger;
} TriggerData;

tg_event
    describes event for which the function is called. You may use the
    following macros to examine tg_event:

    TRIGGER_FIRED_BEFORE(event) returns TRUE if trigger fired BEFORE;
    TRIGGER_FIRED_AFTER(event) returns TRUE if trigger fired AFTER;
    TRIGGER_FIRED_FOR_ROW(event) returns TRUE if trigger fired for
                                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 pointer to structure describing the triggered relation. Look at `src/include/utils/rel.h` for details about this structure. The most interesting 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;
    FmgrInfo    tgfunc;
    int16       tgtype;
    bool        tgenabled;
    bool        tgisconstraint;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgattr[FUNC_MAX_ARGS];
    char        **tgargs;
} Trigger;
```

`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.

## Visibility of Data Changes

Postgres 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 which 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!

## Examples

There are more complex examples in 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 */

HeapTuple          trigf(void);

HeapTuple
trigf()
{
    TupleDesc      tupdesc;
    HeapTuple      rettuple;
    char           *when;
    bool           checknull = false;
    bool           isnull;
    int            ret, i;

    if (!CurrentTriggerData)
        elog(WARN, "trigf: triggers are not initialized");

    /* tuple to return to Executor */
    if (TRIGGER FIRED BY UPDATE(CurrentTriggerData->tg_event))
        rettuple = CurrentTriggerData->tg_newtuple;
    else
        rettuple = CurrentTriggerData->tg_trigtuple;
```

```

/* check for NULLs ? */
if (!TRIGGER_FIRED_BY_DELETE(CurrentTriggerData->tg_event)
&&
    TRIGGER_FIRED_BEFORE(CurrentTriggerData->tg_event))
    checknull = true;

if (TRIGGER_FIRED_BEFORE(CurrentTriggerData->tg_event))
    when = "before";
else
    when = "after ";

tupdesc = CurrentTriggerData->tg_relation->rd_att;
CurrentTriggerData = NULL;

/* Connect to SPI manager */
if ((ret = SPI_connect()) < 0)
    elog(WARN, "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(WARN, "trigf (fired %s): SPI_exec returned %d",
when, ret);

i = 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)
{
    i = SPI_getbinval(rettuple, tupdesc, 1, &isnull);
    if (isnull)
        rettuple = NULL;
}

return (rettuple);
}

```

Now, compile and create table ttest (x int4); create function trigf () returns opaque as '...path\_to\_so' language 'c';

```

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 15. Server Programming Interface

The *Server Programming Interface* (SPI) gives users the ability to run SQL queries inside user-defined C functions. The available Procedural Languages (PL) give an alternate means to access these capabilities.

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.

SPI procedures are always called by some (upper) Executor and the SPI manager uses the Executor to run your queries. Other procedures may be called by the Executor running queries from your procedure.

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 be changed in future versions.

Other restrictions are the inability to execute BEGIN, END and ABORT (transaction control statements) and cursor operations. 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.

## Interface Functions

### SPI\_connect

#### Name

`SPI_connect` Connects your procedure to the SPI manager.

#### Synopsis

```
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 to the Postgres backend. You should call this function if you will need to execute queries. Some utility SPI functions may be called from un-connected procedures.

You may get `SPI_ERROR_CONNECT` error if `SPI_connect` is called from an already connected procedure - e.g. if you directly call one procedure from another connected one. Actually, while the child procedure will be able to use SPI, your parent procedure will not be able to continue to use SPI after the child returns (if `SPI_finish` is called by the child). It's bad practice.

#### Usage

#### Algorithm

`SPI_connect` performs the following:

Initializes the SPI internal structures for query execution and memory management.

## SPI\_finish

### Name

`SPI_finish` Disconnects your procedure from the SPI manager.

### Synopsis

```
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 Postgres backend. You should call this function after completing operations through the SPI manager.

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! Note that you can safely skip the call to `SPI_finish` if you abort the transaction (via `elog(ERROR)`).

### 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.

## SPI\_exec

### Name

`SPI_exec` Creates an execution plan (parser+planner+optimizer) and executes a query.

### Synopsis

```
SPI_exec(query, tcount)
```

### Inputs

```
char *query
```

String containing query plan

```
int tcount
```

Maximum number of tuples to return

### Outputs

```
int
```

`SPI_OK_EXEC` if properly disconnected

`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. For example,

```
SPI_exec ("insert into table select * from table", 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 many 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` returned and `SPI_processed > 0` then you may use global pointer `SPITupleTable *SPI_tuptable` to access the selected tuples: Also NOTE, that `SPI_finish` frees and makes all `SPITupleTables` unusable! (See Memory management).

`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).

## Algorithm

`SPI_exec` 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.

## SPI\_prepare

### Name

`SPI_prepare` Connects your procedure to the SPI manager.

### Synopsis

```
SPI_prepare(query, nargs, argtypes)
```

### Inputs

*query*

Query string

*nargs*

Number of input parameters (\$1 ... \$nargs - as in SQL-functions)

*argtypes*

Pointer list of type OIDs to input arguments

### 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

`nargs` is number of parameters (\$1 ... \$nargs - as in SQL-functions), and `nargs` may be 0 only if there is not any \$1 in query.

Execution of prepared execution plans is sometimes much faster so this feature may be useful if the same query will be executed many times.

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. See `SPI_saveplan`.

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.

## SPI\_saveplan

### Name

`SPI_saveplan` Saves a passed plan

### Synopsis

`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 Postgres 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 consequent 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.

## SPI\_execp

### Name

`SPI_execp` Executes a plan from `SPI_saveplan`

### Synopsis

```
SPI_execp(plan,
          values,
          nulls,
          tcount)
```

### Inputs

`void *plan`

Execution plan

Datum \**values*

Actual parameter values

char \**nulls*

Array describing what parameters get NULLs

'n' indicates NULL allowed

' ' indicates NULL not allowed

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` 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 Postgres 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 a work around, there is the ability to reuse prepared plans in the consequent invocations of your procedure in the current session. Use `SPI_execp` to execute this saved plan.

## Usage

If `nulls` is NULL then `SPI_execp` assumes that all values (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.

## Interface Support Functions

All functions described below may be used by connected and unconnected procedures.

### SPI\_copytuple

#### Name

`SPI_copytuple` Makes copy of tuple in upper Executor context

#### Synopsis

`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. See the section on Memory Management.

#### Usage

TBD

## SPI\_modifytuple

### Name

`SPI_modifytuple` Modifies tuple of relation

### Synopsis

```
SPI_modifytuple(rel, tuple , nattrs
, attnum , Values , Nulls)
```

### Inputs

Relation *rel*

HeapTuple *tuple*

Input tuple to be modified

int *nattrs*

Number of attribute numbers in *attnum*

int \* *attnum*

Array of numbers of the attributes which are to be changed

Datum \* *Values*

New values for the attributes specified

char \* *Nulls*

Which attributes 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` Modifies a tuple in upper Executor context. See the section on Memory Management.

## Usage

If successful, a pointer to the new tuple is returned. The new tuple is allocated in upper Executor context (see Memory management). Passed tuple is not changed.

## SPI\_fnumber

### Name

`SPI_fnumber` Finds the attribute number for specified attribute

### Synopsis

```
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.

## SPI\_fname

### Name

SPI\_fname Finds the attribute name for the specified attribute

### Synopsis

SPI\_fname(*tupdesc*, *fnumber*)

### Inputs

TupleDesc *tupdesc*

Input tuple description

char \* *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.

## SPI\_getvalue

### Name

`SPI_getvalue` Returns the string value of the specified attribute

### Synopsis

```
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

Allocates memory as required by the value.

## SPI\_getbinval

### Name

`SPI_getbinval` Returns the binary value of the specified attribute

### Synopsis

```
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 binary value of the specified attribute.

### Usage

Attribute numbers are 1 based.

### Algorithm

Does not allocate new space for the binary value.

## SPI\_gettype

### Name

`SPI_gettype` Returns the type name of the specified attribute

### Synopsis

```
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.

### Usage

Attribute numbers are 1 based.

### Algorithm

Does not allocate new space for the binary value.

## SPI\_gettypeid

### Name

`SPI_gettypeid` Returns the type OID of the specified attribute

### Synopsis

```
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.

### Algorithm

TBD

## SPI\_getrelname

### Name

`SPI_getrelname` Returns the name of the specified relation

### Synopsis

```
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.

### Usage

TBD

### Algorithm

Copies the relation name into new storage.

## SPI\_palloc

### Name

`SPI_palloc` Allocates memory in upper Executor context

### Synopsis

```
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. See section on memory management.

### Usage

TBD

## SPI\_realloc

### Name

`SPI_realloc` Re-allocates memory in upper Executor context

### Synopsis

```
SPI_realloc(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_realloc` re-allocates memory in upper Executor context. See section on memory management.

### Usage

TBD

## SPI\_pfree

### Name

`SPI_pfree` Frees memory from upper Executor context

### Synopsis

```
SPI_pfree(pointer)
```

### Inputs

```
void * pointer
```

Pointer to existing storage

### Outputs

None

### Description

`SPI_pfree` frees memory in upper Executor context. See section on memory management.

### Usage

TBD

## Memory Management

Server allocates memory in memory contexts in such way that allocations made in one context may be freed by context destruction without affecting allocations made in other contexts. All allocations (via `palloc`, etc) are made in the context which are chosen as current one. You'll get unpredictable results if you'll try to free (or reallocate) memory allocated not in current context.

Creation and switching between memory contexts are subject of SPI manager memory management.

SPI procedures deal with two memory contexts: upper Executor memory context and procedure memory context (if connected).

Before a procedure is connected to the SPI manager, current memory context is upper Executor context so all allocation made by the procedure itself via `palloc/repalloc` or by SPI utility functions before connecting to SPI are made in this context.

After `SPI_connect` is called current context is the procedure's one. All allocations made via `palloc/repalloc` or by SPI utility functions (except for `SPI_copytuple`, `SPI_modifytuple`, `SPI_palloc` and `SPI_repalloc`) 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!

If you want to return something to the upper Executor then you have to allocate memory for this in the upper context!

SPI has no ability to automatically free allocations in the upper Executor context!

SPI automatically frees memory allocated during execution of a query when this query is done!

## Visibility of Data Changes

Postgres 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.

Changes made by query Q are visible by queries which are started after query Q, no matter whether they are started inside Q (during the execution of Q) or after Q is done.

## Examples

This example of SPI usage demonstrates the visibility rule. There are more complex examples in 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)
{
    int ret;
    int proc = 0;

    SPI_connect();

    ret = SPI_exec(textout(sql), 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;

        for (ret = 0; ret < proc; ret++)
```

```

        {
            HeapTuple tuple = tupdesc->vals[ret];

            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();

    return (proc);
}

```

Now, compile and create the function:

```
create function execq (text, int4) returns int4 as '...path_to_so'
language 'c';
```

```
vac=> select execq('create table a (x int4)', 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

```

# Chapter 16. Large Objects

In Postgres, data values are stored in tuples and individual tuples cannot span data pages. Since the size of a data page is 8192 bytes, the upper limit on the size of a data value is relatively low. To support the storage of larger atomic values, Postgres provides a large object interface. This interface provides file oriented access to user data that has been declared to be a large type. This section describes the implementation and the programming and query language interfaces to Postgres large object data.

## Historical Note

Originally, Postgres 4.2 supported three standard implementations of large objects: as files external to Postgres, as external files managed by Postgres, and as data stored within the Postgres database. It causes considerable confusion among users. As a result, we only support large objects as data stored within the Postgres database in PostgreSQL. Even though it is slower to access, it provides stricter data integrity. For historical reasons, this storage scheme is referred to as Inversion large objects. (We will use Inversion and large objects interchangeably to mean the same thing in this section.)

## Implementation Features

The Inversion 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.

## Interfaces

The facilities Postgres 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 Postgres v6.5, though it has been an implicit requirement in previous versions, resulting in misbehavior if ignored.

The Postgres 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. Postgres provides a set of routines that support opening, reading, writing, closing, and seeking on large objects.

## Creating a Large Object

The routine

```
Oid lo_creat(PGconn *conn, int mode)
```

creates a new large object. *mode* is a bitmask describing several different attributes of the new object. The symbolic constants listed here are defined in

`$PGROOT/src/backend/libpq/libpq-fs.h`. The access type (read, write, or both) is controlled by OR ing together the bits `INV_READ` and `INV_WRITE`. If the large object should be archived -- that is, if historical versions of it should be moved periodically to a special archive relation -- then the `INV_ARCHIVE` bit should be set. The low-order sixteen bits of mask are the storage manager number on which the large object should reside. For sites other than Berkeley, these bits should always be zero. The commands below create an (Inversion) large object:

```
inv_oid = lo_creat(INV_READ | INV_WRITE | INV_ARCHIVE);
```

## Importing a Large Object

To import a UNIX file as a large object, call

```
Oid lo_import(PGconn *conn, const char *filename)
```

*filename* specifies the Unix pathname of the file to be imported as a large object.

## Exporting a Large Object

To export a large object into UNIX file, call

```
int lo_export(PGconn *conn, Oid loObjId, const char *filename)
```

The *loObjId* argument specifies the Oid of the large object to export and the *filename* argument specifies the UNIX pathname of the file.

## Opening an Existing Large Object

To open an existing large object, call

```
int lo_open(PGconn *conn, Oid loObjId, int mode)
```

The *loObjId* argument specifies the Oid of the large object to open. The mode bits control whether the object is opened for reading (`INV_READ`), writing 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`.

## 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.

## 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.

## 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`.

## 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.

## Built in registered 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';
```

## Accessing Large Objects from LIBPQ

Below 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 readers

benefit. This program can be found in `../src/test/examples`. 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.

## Sample Program

```

/*-----
 *
 * testlo.c--
 *   test using large objects with libpq
 *
 * Copyright (c) 1994, Regents of the University of California
 *
 *
 * IDENTIFICATION
 *   /usr/local/devel/pglite/cvs/src/doc/manual.me,v 1.16
 *
 *-----
 */
#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;
    }
    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;
    }
    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 17. libpq - C Library

`libpq` is the C application programmer's interface to Postgres. `libpq` is a set of library routines that allow client programs to pass queries to the Postgres backend server and to receive the results of these queries. `libpq` is also the underlying engine for several other Postgres application interfaces, including `libpq++` (C++), `libpq Tcl` (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/regress
../src/test/examples
../src/bin/psql
```

Frontend programs which use `libpq` must include the header file `libpq-fe.h` and must link with the `libpq` library.

## Database Connection Functions

The following routines deal with making a connection to a Postgres 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 non-blocking analogues `PQconnectStart` / `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 a null value or a value containing spaces, surround it with single quotes, e.g., `keyword = 'a value'`. Single quotes within the value must be written as `\'`. Spaces around the equal sign are optional.) The currently recognized parameter keywords are:

`host`

Name of host to connect to. If a non-zero-length string is specified, TCP/IP communication is used. Using this parameter causes a hostname look-up. See `hostaddr`.

`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 non-zero-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 hostname look-up 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 which is not the name of the machine at `hostaddr`.

Without both a host name and host address, `libpq` will connect using a local Unix domain socket.

`port`

Port number to connect to at the server host, or socket filename 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.

If any parameter is unspecified, then the corresponding environment variable (see "Environment Variables" section) 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 non-blocking 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 postmaster.

`CONNECTION_AUTH_OK`: Received authentication; waiting for backend startup.

`CONNECTION_SETENV`: Negotiating environment.

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.

`PQconnectPoll` will currently block under Windows, unless `libpq` is compiled with `WIN32_NON_BLOCKING_CONNECTIONS` defined. This code has not yet been

tested under Windows, and so it is currently off by default. This may be changed in the future.

These functions leave the socket in a non-blocking state as if `PQsetnonblocking` had been called.

`PQconnndefaults` Returns the default connection options.

```
PQconninfoOption *PQconnndefaults(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 `PQconnndefaults()`.

In Postgres versions before 7.0, `PQconnndefaults()` returned a pointer to a static array, rather than a dynamically allocated array. That wasn't 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 postmaster, 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 non-blocking 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 postmaster, 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 non-blocking 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 Postgres 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!

## Query 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.

`PQexec` Submit a query to Postgres 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 query 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 query 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 Postgres release 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 query.

```
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 tuples returned by the query. Note that a `SELECT` that happens to retrieve zero tuples still shows `PGRES_TUPLES_OK`. `PGRES_COMMAND_OK` is for commands that can never return tuples (`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.

`PQntuples` Returns the number of tuples (instances) in the query result.

```
int PQntuples(const PGresult *res);
```

`PQnfields` Returns the number of fields (attributes) in each tuple of the query result.

```
int PQnfields(const PGresult *res);
```

`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.

`PQfname` Returns the field (attribute) name associated with the given field index. Field indices start at 0.

```
char *PQfname(const PGresult *res,
              int field_index);
```

`PQfnumber` Returns the field (attribute) 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_num);
```

You can query the system table `pg_type` to obtain the name and properties of the various datatypes. The OIDs of the built-in datatypes are defined in `src/include/catalog/pg_type.h` in the source tree.

`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.

**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);
```

**PQgetvalue** Returns a single field (attribute) value of one tuple 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 ASCII 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.

**PQgetlength** Returns the length of a field (attribute) 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 ASCII-represented values, this size has little to do with the binary size reported by **PQfsize**.

**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.)

**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 tuple inserted, if the SQL command was an INSERT. 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 tuple inserted, if the SQL command was an INSERT. Otherwise, returns an empty string.

```
char * PQoidStatus(const PGresult *res);
```

This function is deprecated in favor of PQoidValue and is not thread-safe.

**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.

**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.

## Asynchronous Query Processing

The `PQexec` function is adequate for submitting queries in simple synchronous applications. It has a couple of major deficiencies however:

`PQexec` waits for the query 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 query. (It can be done from a signal handler, but not otherwise.)

`PQexec` can return only one `PGresult` structure. If the submitted query 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 non-blocking connection to the backend.

`PQsetnonblocking` Sets the state of the connection to non-blocking.

```
int PQsetnonblocking(PGconn *conn)
```

this function will ensure that 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 non-blocking 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 TRUE if the connection is set to non-blocking mode, FALSE if blocking.

`PQsendQuery` Submit a query to Postgres without waiting for the result(s). TRUE is returned if the query was successfully dispatched, FALSE 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 query results. `PQsendQuery` may not be called again (on the same connection) until `PQgetResult` has returned NULL, indicating that the query 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 query is done. (If called when no query 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 query 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 query 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(2)` 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 non-blocking 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  $\geq 0$ ; 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(2)`. This allows an application using a blocking connection to wait for either backend responses or other conditions. If the result of `select(2)` 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.

Non-blocking 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(2)` 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 "Asynchronous Notification", below).

A frontend that uses `PQsendQuery/PQgetResult` can also attempt to cancel a query that is still being processed by the backend.

`PQrequestCancel` Request that Postgres abandon processing of the current query.

```
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 query will terminate early and return an error result. If the cancellation fails (say, because the backend was already done processing the query), then there will be no visible result at all.

Note that if the current query 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 query.

## Fast Path

Postgres 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.

## Asynchronous Notification

Postgres 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 queries. 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 Postgres 6.4 and later, the `be_pid` is the notifying backend's, whereas in earlier versions it was always your own backend's PID.

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(2)` 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.

## Functions Associated with the COPY Command

The COPY command in Postgres 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(3)`, this routine copies up to `length-1` characters into `string`. It is like `gets(3)`, however, in that it converts the terminating newline into a null character. `PQgetline` returns EOF at EOF, 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` Syncs 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 `PQgetline`. 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 query. 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 float8)");
PQexec(conn, "copy foo from stdin");
PQputline(conn, "3\tthehello 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 query string.

## 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)
```

## 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`.

## 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 a non-zero-length string is specified, TCP/IP communication is used. Without a host name, libpq will connect using a local Unix domain socket.

`PGPORT` sets the default port or local Unix domain socket file extension for communicating with the Postgres backend.

`PGDATABASE` sets the default Postgres database name.

`PGUSER` sets the username used to connect to the database and for authentication.

`PGPASSWORD` sets the password used if the backend demands password authentication.

`PGREALM` sets the Kerberos realm to use with Postgres, if it is different from the local realm. If `PGREALM` is set, Postgres 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 Postgres 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 Postgres 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 Postgres).

The following environment variables can be used to specify default internal behavior for every Postgres session:

`PGGEOQO` sets the default mode for the genetic optimizer.

Refer to the `SET SQL` command for information on correct values for these environment variables.

## Threading Behavior

libpq is thread-safe as of Postgres 7.0, so long as no two threads attempt to manipulate the same `PGconn` object at the same time. In particular, you can't 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 multi-thread programs. `PQoidStatus` can be replaced by `PQoidValue`. There is no good reason to call `fe_setauthsvc` at all.

## Sample Programs

### Sample Program 1

```

/*
 * testlibpq.c Test the C version of Libpq, the Postgres 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 instances 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 instances */
    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); */
}

```

## Sample 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,
received\n",
                    notify->relname, notify->be_pid);
            free(notify);
        }
    }

    /* close the connection to the database and cleanup */
    PQfinish(conn);
}

```

### Sample Program 3

```

/*
 * testlibpq3.c Test the C version of Libpq, the Postgres frontend
 * library. tests the binary cursor interface
 *
 *
 *
 * populate a database by doing the following:
 *
 * CREATE TABLE test1 (i int4, d float4, p polygon);
 *
 * INSERT INTO test1 values (1, 3.567, '(3.0, 4.0, 1.0,

```

```

* 2.0)::polygon);
*
* INSERT INTO test1 values (2, 89.05, '(4.0, 3.0, 2.0,
* 1.0)::polygon);
*
* the expected output is:
*
* tuple 0: got i = (4 bytes) 1, d = (4 bytes) 3.567000, p = (4
* bytes) 2 points  boundingbox = (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  boundingbox =
* (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 instances 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 VARHDRSZ
         */
        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 \tboundingbox = (hi=%f/%f, lo
= %f,%f)\n",
            PQgetlength(res, i, d_fnum),
            pval->npts,
            pval->boundingbox.xh,
            pval->boundingbox.yh,
            pval->boundingbox.xl,
            pval->boundingbox.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);  
}
```

# Chapter 18. libpq - C++ Binding Library

`libpq++` is the C++ API to Postgres. `libpq++` is a set of classes which allow client programs to connect to the Postgres 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 to the Postgres 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 Postgres 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. Three short programs are listed at the end of this section as examples of `libpq++` programming (though not necessarily of good programming). There are several examples of `libpq++` applications in `src/libpq++/examples`, including the source code for the three examples in this chapter.

## Control and Initialization

### 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 the *libpq - C Library* 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 `PQconnectdb` `conninfo` style strings.

`PGHOST` sets the default server name. If a non-zero-length string is specified, TCP/IP communication is used. Without a host name, `libpq` will connect using a local Unix domain socket.

`PGPORT` sets the default port or local Unix domain socket file extension for communicating with the Postgres backend.

`PGDATABASE` sets the default Postgres database name.

`PGUSER` sets the username used to connect to the database and for authentication.

`PGPASSWORD` sets the password used if the backend demands password authentication.

PGREALM sets the Kerberos realm to use with Postgres, if it is different from the local realm. If PGREALM is set, Postgres 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 Postgres 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 Postgres 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 Postgres 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.

## libpq++ Classes

### Connection Class: `PgConnection`

The connection class makes the actual connection to the database and is inherited by all of the access classes.

### 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.

## Database Connection Functions

`PgConnection` makes a new connection to a backend database server.

```
PgConnection::PgConnection(const char *conninfo)
```

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.

```
int PgConnection::ConnectionBad()
```

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() for details.

## Query Execution Functions

Exec Sends a query 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 of the query. The following status results can be expected:

PGRES\_EMPTY\_QUERY

PGRES\_COMMAND\_OK, if the query was a command

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 command query to the backend server.

```
int PgConnection::ExecCommandOk(const char *query)
```

Returns TRUE if the command query succeeds.

**ExecTuplesOk** Sends a command query to the backend server.

```
int PgConnection::ExecTuplesOk(const char *query)
```

Returns TRUE if the command query succeeds.

**ErrorMessage** Returns the last error message text.

```
const char *PgConnection::ErrorMessage()
```

**Tuples** Returns the number of tuples (instances) in the query result.

```
int PgDatabase::Tuples()
```

**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()
```

**Fields** Returns the number of fields (attributes) in each tuple of the query result.

```
int PgDatabase::Fields()
```

**FieldName** Returns the field (attribute) name associated with the given field index. Field indices start at 0.

```
const char *PgDatabase::FieldName(int field_num)
```

**FieldNum PQnumber** Returns the field (attribute) index associated with the given field name.

```
int PgDatabase::FieldNum(const char* field_name)
```

-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)
```

**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)
```

`FieldSize` Returns the size in bytes of the field associated with the given field index. Field indices start at 0.

```
short PgDatabase::FieldSize(int field_num)
```

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.

```
short PgDatabase::FieldSize(const char *field_name)
```

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.

`GetValue` Returns a single field (attribute) value of one tuple of a PGresult. Tuple and field indices start at 0.

```
const char *PgDatabase::GetValue(int tup_num, int field_num)
```

For most queries, the value returned by `GetValue` is a null-terminated ASCII 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 (attribute) 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)
```

For most queries, the value returned by `GetValue` is a null-terminated ASCII 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 (attribute) in bytes. Tuple and field indices start at 0.

```
int PgDatabase::GetLength(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 `GetValue`. Note that for ASCII-represented values, this size has little to do with the binary size reported by `PQfsize`.

`GetLength` Returns the length of a field (attribute) in bytes. Tuple and field indices start at 0.

```
int PgDatabase::GetLength(int tup_num, const char* field_name)
```

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 ASCII-represented values, this size has little to do with the binary size reported by `PQfsize`.

`DisplayTuples` Prints out all the tuples and, optionally, the attribute names to the specified output stream.

```
void PgDatabase::DisplayTuples(FILE *out = 0, int fillAlign
= 1,
    const char* fieldSep = "|",int printHeader = 1, int quiet
= 0)
```

`PrintTuples` Prints out all the tuples and, optionally, the attribute names to the specified output stream.

```
void PgDatabase::PrintTuples(FILE *out = 0, int printAttName =
1,
    int terseOutput = 0, int width = 0)
```

`GetLine`

```
int PgDatabase::GetLine(char* string, int length)
```

`PutLine`

```
void PgDatabase::PutLine(const char* string)
```

`OidStatus`

```
const char *PgDatabase::OidStatus()
```

`EndCopy`

```
int PgDatabase::EndCopy()
```

## Asynchronous Notification

Postgres supports asynchronous notification via the **LISTEN** and **NOTIFY** commands. A backend registers its interest in a particular semaphore with the **LISTEN** command. All backends that are listening on a particular named semaphore 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 the relation.

**Note:** In the past, the documentation has associated the names used for asynchronous notification with relations or classes. However, there is in fact no direct linkage of the two concepts in the implementation, and the named semaphore in fact does not need to have a corresponding relation previously defined.

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 query, 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 is 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.

## Functions Associated with the COPY Command

The **copy** command in Postgres 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 (3)`, this routine copies up to *length*-1 characters into *string*. It is like `gets (3)`, however, in that it converts the terminating newline into a null character.

`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 single 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 a single period character (".") to indicate to the backend that it has finished sending its data.

`PgDatabase::EndCopy` syncs 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 query.

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 float8)");
data.Exec("copy foo from stdin");
data.putline("3\etHello World\et4.5\en");
data.putline("4\etGoodbye World\et7.11\en");
&...
data.putline(".\en");
data.endcopy();
```

# Chapter 19. pgtcl - TCL Binding Library

`pgtcl` is a tcl package for front-end programs to interface with Postgres backends. It makes most of the functionality of `libpq` available to tcl scripts.

This package was originally written by Jolly Chen.

## Commands

Table 19-1. `pgtcl` Commands

Command	Description
<code>pg_connect</code>	opens a connection to the backend server
<code>pg_disconnect</code>	closes a connection
<code>pg_conndefaults</code>	get connection options and their defaults
<code>pg_exec</code>	send a query to the backend
<code>pg_result</code>	manipulate the results of a query
<code>pg_select</code>	loop over the result of a SELECT statement
<code>pg_listen</code>	establish a callback for NOTIFY messages
<code>pg_lo_creat</code>	create a large object
<code>pg_lo_open</code>	open a large object
<code>pg_lo_close</code>	close a large object
<code>pg_lo_read</code>	read a large object
<code>pg_lo_write</code>	write a large object
<code>pg_lo_lseek</code>	seek to a position in a large object
<code>pg_lo_tell</code>	return the current seek position of a large object
<code>pg_lo_unlink</code>	delete a large object
<code>pg_lo_import</code>	import a Unix file into a large object
<code>pg_lo_export</code>	export a large object into a Unix file

These commands are described further on subsequent pages.

The `pg_lo*` routines are interfaces to the Large Object features of Postgres. 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/END 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/END` transaction block.

## Examples

Here's a small example of how to use the routines:

```
# 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
}
```

## pgtcl Command Reference Information

### pg\_connect

#### Name

`pg_connect` opens a connection to the backend server

#### Synopsis

```
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.

#### 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 Postgres 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

(Not yet documented)

## pg\_disconnect

### Name

`pg_disconnect` closes a connection to the backend server

### Synopsis

`pg_disconnect dbHandle`

### Inputs

*dbHandle*

Specifies a valid database handle.

### Outputs

None

### Description

`pg_disconnect` closes a connection to the Postgres backend.

## pg\_conndefaults

### Name

`pg_conndefaults` obtain information about default connection parameters

### Synopsis

`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`

## pg\_exec

### Name

`pg_exec` send a query string to the backend

### Synopsis

```
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 Postgres 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`.

## pg\_result

### Name

`pg_result` get information about a query result

### Synopsis

`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.

`-list VarName`

assign the results to a list of lists.

`-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.

## pg\_select

### Name

`pg_select` loop over the result of a SELECT statement

### Synopsis

```
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

*resultHandle*

the return result is either an error message or a handle for a query result.

### Description

`pg_select` submits a SELECT query to the Postgres 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.

### 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)]
}
```

## pg\_listen

### Name

`pg_listen` sets or changes a callback for asynchronous NOTIFY messages

### Synopsis

```
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 Postgres 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 Postgres 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`.

## pg\_lo\_creat

### Name

`pg_lo_creat` create a large object

### Synopsis

`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`, `INV_WRITE`, and `INV_ARCHIVE`.  
The OR delimiter character is "|".

```
[pg_lo_creat $conn "INV_READ|INV_WRITE" ]
```

## pg\_lo\_open

### Name

`pg_lo_open` open a large object

### Synopsis

`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".

## pg\_lo\_close

### Name

`pg_lo_close` close a large object

### Synopsis

```
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

## pg\_lo\_read

### Name

`pg_lo_read` read a large object

### Synopsis

```
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.**pg\_lo\_write****Name**`pg_lo_write` write a large object**Synopsis**`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.

## pg\_lo\_lseek

### Name

`pg_lo_lseek` seek to a position in a large object

### Synopsis

```
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".

## pg\_lo\_tell

### Name

`pg_lo_tell` return the current seek position of a large object

### Synopsis

```
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

## pg\_lo\_unlink

### Name

`pg_lo_unlink` delete a large object

### Synopsis

```
pg_lo_unlink conn lobjId
```

### Inputs

*conn*

Specifies a valid database connection.

*lobjId*

Identifier for a large object. XXX Is this the same as objOid in other calls?? - thomas 1998-01-11

## Outputs

None

## Description

`pg_lo_unlink` deletes the specified large object.

## Usage

# pg\_lo\_import

## Name

`pg_lo_import` import a large object from a Unix file

## Synopsis

`pg_lo_import conn filename`

## Inputs

*conn*

Specifies a valid database connection.

*filename*

Unix file name.

## Outputs

None XXX Does this return a lobjId? Is that the same as the objOid in other calls? thomas - 1998-01-11

## 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.

## pg\_lo\_export

### Name

`pg_lo_export` export a large object to a Unix file

### Synopsis

```
pg_lo_export conn lobjId filename
```

### Inputs

*conn*

Specifies a valid database connection.

*lobjId*

Large object identifier. XXX Is this the same as the objOid in other calls?? thomas - 1998-01-11

*filename*

Unix file name.

### Outputs

None XXX Does this return a lobjId? Is that the same as the objOid in other calls? thomas - 1998-01-11

### 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 20. libpgeasy - Simplified C Binding Library

**Author:** Written by Bruce Momjian (root@candle.pha.pa.us (mailto:root@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. It consists of set of simplified C functions that encapsulate the functionality of libpq. The functions are:

```
PGresult *doquery(char *query);
PGconn *connectdb();
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 can not 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 a variety of demonstration programs in the source directory.

# Chapter 21. ecpg - Embedded SQL in C

This describes an embedded SQL in C package for Postgres. It is written by Linus Tolke () and Michael Meskes ().

**Note:** Permission is granted to copy and use in the same way as you are allowed to copy and use the rest of PostgreSQL.

## Why Embedded SQL?

Embedded SQL has some small advantages over other ways to handle SQL queries. It takes care of all the tedious moving of information to and from variables in your 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 meet this standard as much as possible. So it is possible to port programs with embedded SQL written for other RDBMS packages to Postgres and thus promoting the spirit of free software.

## The Concept

You write your program in C with some special SQL things. For declaring variables that can 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 variables that are used as input to the SQL statements and variables that will contain the result are passed.

Then you compile and at link time you link with a special library that contains the functions used. These functions (actually it is mostly one single function) fetches the information from the arguments, performs the SQL query using the ordinary interface (`libpq`) and puts back the result in the arguments dedicated for output.

Then you run your program and when the control arrives to the SQL statement the SQL statement is performed against the database and you can continue with the result.

## How To Use ecpg

This section describes how to use the ecpg tool.

### Preprocessor

The preprocessor is called ecpg. After installation it resides in the Postgres `bin/` directory.

## Library

The *ecpg* library is called `libecpg.a` or `libecpg.so`. Additionally, the library uses the `libpq` library for communication to the Postgres server so you will have to link your program with `-lecpg -lpq`.

The library has some methods that are "hidden" but that could prove very useful sometime.

`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 logs its arguments and result.

The most important one (`ECPGdo`) that is called on almost all SQL statements logs both its expanded string, i.e. the string with all the input variables inserted, and the result from the Postgres 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.

## Error handling

To be able to detect errors from the Postgres server you 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 following:

```
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 sqltext[8];
} sqlca;

```

If an error occurred in the last SQL statement then `sqlca.sqlcode` will be non-zero. If `sqlca.sqlcode` is less than 0 then this is some kind of serious error, like the database definition does not match the query given. If it is bigger than 0 then this 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.

List of errors that can occur:

-12, Out of memory in line %d.

Does not normally occur. This is a sign that your virtual memory is exhausted.

-200, Unsupported type %s on line %d.

Does not normally occur. This is a sign that 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, Too many arguments line %d.

This means that Postgres 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, Too few arguments line %d.

This means that Postgres has returned fewer arguments than we have host variables. Perhaps you have too many host variables in the **INTO :var1,:var2**-list.

-203, Too many matches line %d.

This means that the query has returned several lines but the variables specified are no arrays. The **SELECT** you made probably was not unique.

-204, Not correctly formatted int type: %s line %d.

This means that the host variable is of an int type and the field in the Postgres database is of another type and contains a value that cannot be interpreted as an int. The library uses `strtol` for this conversion.

-205, Not correctly formatted unsigned type: %s line %d.

This means that the host variable is of an unsigned int type and the field in the Postgres 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, Not correctly formatted floating point type: %s line %d.

This means that the host variable is of a float type and the field in the Postgres database is of another type and contains a value that cannot be interpreted as a float. The library uses `strtod` for this conversion.

-207, Unable to convert %s to bool on line %d.

This means that the host variable is of a bool type and the field in the Postgres database is neither 't' nor 'f'.

-208, Empty query line %d.

Postgres returned `PGRES_EMPTY_QUERY`, probably because the query indeed was empty.

-220, No such connection %s in line %d.

The program tries to access a connection that does not exist.

-221, Not connected in line %d.

The program tries to access a connection that does exist but is not open.

-230, Invalid statement name %s in line %d.

The statement you are trying to use has not been prepared.

-400, Postgres error: %s line %d.

Some Postgres error. The message contains the error message from the Postgres backend.

-401, Error in transaction processing line %d.

Postgres signalled to us that we cannot start, commit or rollback the transaction.

-402, connect: could not open database %s.

The connect to the database did not work.

100, Data not found line %d.

This is a "normal" error that tells you that what you are querying cannot be found or we have gone through the cursor.

## Limitations

What will never be included and why or what cannot be done with this concept.

Oracle's single tasking possibility

Oracle version 7.0 on AIX 3 uses the OS-supported locks on the shared memory segments and allows the application designer to link an application in a so called single tasking way. Instead of starting one client process per application process both

the database part and the application part is run in the same process. In later versions of Oracle this is no longer supported.

This would require a total redesign of the Postgres access model and that effort can not justify the performance gained.

## Porting From Other RDBMS Packages

The design of *ecpg* follows SQL standard. So porting from a standard RDBMS should not be a problem. Unfortunately there is no such thing as a standard RDBMS. So *ecpg* also tries to understand syntax additions 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 Michael Meskes (). Note, however, that we list only incompatibilities from a precompiler of another RDBMS to *ecpg* and not additional *ecpg* features that these RDBMS do not have.

Syntax of `FETCH` command

The standard syntax of the `FETCH` command is:

```
FETCH [direction] [amount] IN|FROM cursor name.
```

ORACLE, however, does not use the keywords `IN` resp. `FROM`. This feature cannot be added since it would create parsing conflicts.

## Installation

Since version 0.5 *ecpg* is distributed together with Postgres. So you should get your precompiler, libraries and header files compiled and installed by default as a part of your installation.

## For the Developer

This section is for those who want to develop the *ecpg* interface. It describes how the things work. The ambition is to make this section contain things for those that want to have a look inside and the section on How to use it should be enough for all normal questions. So, read this before looking at the internals of the *ecpg*. If you are not interested in how it really works, skip this section.

## ToDo List

This version the preprocessor has some flaws:

Library functions

`to_date` et al. do not exist. But then Postgres has some good conversion routines itself. So you probably won't miss these.

### Structures and unions

Structures and unions have to be defined in the declare section.

### Missing statements

The following statements are not implemented thus far:

```
exec sql allocate
```

```
exec sql deallocate
```

```
SQLSTATE
```

```
message 'no data found'
```

The error message for "no data" in an `exec sql insert select from` statement has to be 100.

```
sqlwarn[6]
```

`sqlwarn[6]` should be 'W' if the `PRECISION` or `SCALE` value specified in a `SET DESCRIPTOR` statement will be ignored.

## The Preprocessor

The first four lines written to the output are constant additions by `ecpg`. These are two comments and two include lines necessary for the interface to the library.

Then the preprocessor works in one pass only, reading the input file and writing to the output as it goes along. Normally it just echoes everything to the output without looking at it further.

When it comes to an **EXEC SQL** statements it intervenes and changes them depending on what it is. The **EXEC SQL** statement can be one of these:

### Declare sections

Declare sections begins with

```
exec sql begin declare section;
```

and ends with

```
exec sql end declare section;
```

In the section only variable declarations are allowed. Every variable declare within this section is also entered in a list of variables indexed on their name together with the corresponding type.

In particular the definition of a structure or union also has to be listed inside a declare section. Otherwise *ecpg* cannot handle these types since it simply does not know the definition.

The declaration is echoed to the file to make the variable a normal C-variable also.

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* itself. So the contents of the specified file is 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 the *password*. Each 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 ignore and not copied from the output.

**Commit statement**

A commit statement looks like

```
exec sql commit;
```

and is translated on the output to

```
ECPGcommit(__LINE__);
```

**Rollback statement**

A rollback statement looks like

```
exec sql rollback;
```

and is translated on the output to

```
ECPGrollback(__LINE__);
```

**Other statements**

Other SQL statements are other statements that start with **exec sql** and ends with **;**. Everything inbetween is treated as an SQL statement and parsed for variable substitution.

Variable substitution occur when a symbol starts with a colon (:). Then a variable with that name is looked for among the variables that were previously declared within a declare section and depending on the variable being for input or output the pointers to the variables are written to the output to allow for access by the function.

For every variable that is part of the SQL request the function gets another ten 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.

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)

## 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 that the preprocessor can do.)

## The Library

The most important function in the library is the `ECPGdo` function. It takes a variable amount of arguments. Hopefully we will not run into machines with limits on the amount of variables that can be accepted by a `vararg` function. This could easily add up to 50 or so arguments.

The arguments are:

A line number

This is a line number for the original line used in error messages only.

#### A string

This is the SQL request that is to be issued. This request is modified by the input variables, i.e. the variables that were not known at compile time but are to be entered in the request. 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.

All the SQL statements are performed in one transaction unless you issue a commit transaction. To get this auto-transaction going the first statement or the first after statement after a commit or rollback always begins a transaction. To disable this feature per default use the `-t` option on the commandline.

To be completed: entries describing the other entries.

# Chapter 22. ODBC Interface

**Note:** Background information originally by Tim Goeke  
(mailto:tgoeke@xpressway.com)

ODBC (Open Database Connectivity) is an abstract API which allows you to write applications which 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 which are transportable between servers from different manufacturers..

## Background

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 Postgres RDBMS.

The backend access come from ODBC drivers, or vendor specific drivers that allow data access. psqLODBC 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 Postgres servers which 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 app).

## 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.

## Writing Applications

If I write an application for Postgres can I write it using ODBC calls to the Postgres 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 VC++ docs.

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.

## Unix Installation

ApplixWare has an ODBC database interface supported on at least some platforms. ApplixWare v4.4.2 has been demonstrated under Linux with Postgres v7.0 using the psqLODBC driver contained in the Postgres distribution.

## Building the Driver

The first thing to note about the psqLODBC driver (or any ODBC driver) is that there must exist a driver manager on the system where the ODBC driver is to be used. There exists a freeware ODBC driver for Unix called iodbc which can be obtained from various locations on the Net, including at AS200 (<http://www.as220.org/FreeODBC/iodbc-2.12.shar.Z>). Instructions for installing iodbc are beyond the scope of this document, but there is a README that can be found inside the iodbc compressed .shar file that should explain how to get it up and running.

Having said that, any driver manager that you can find for your platform should support the psqLODBC driver or any ODBC driver.

The Unix configuration files for psqLODBC have recently been extensively reworked to allow for easy building on supported platforms as well as to allow for support of other Unix platforms in the future. The new configuration and build files for the driver should make it a simple process to build the driver on the supported platforms. Currently these include Linux and FreeBSD but we are hoping other users will contribute the necessary information to quickly expand the number of platforms for which the driver can be built.

There are actually two separate methods to build the driver depending on how you received it and these differences come down to only where and how to run configure and make. The driver can be built in a standalone, client-only installation, or can be built as a part of the main Postgres distribution. The standalone installation is convenient if you have ODBC client applications on multiple, heterogeneous platforms. The integrated installation is convenient when the target client is the same as the server, or when the client and server have similar runtime configurations.

Specifically if you have received the psqLODBC driver as part of the Postgres distribution (from now on referred to as an "integrated" build) then you will configure and make the ODBC driver from the top level source directory of the Postgres distribution along with the rest of its libraries. If you received the driver as a standalone package than you will run configure and make from the directory in which you unpacked the driver source.

### Integrated Installation

This installation procedure is appropriate for an integrated installation.

1. Specify the `--with-odbc` command-line argument for `src/configure`:

```
% ./configure --with-odbc
% make
```

2. Rebuild the Postgres distribution:

```
% make install
```

3. Install the ODBC catalog extensions available in

```
PGROOT/contrib/odbc/odbc.sql:
```

```
% psql -e template1 < $PGROOT/contrib/odbc/odbc.sql
```

where specifying `template1` as the target database will ensure that all subsequent new databases will have these same definitions.

Once configured, the ODBC driver will be built and installed into the areas defined for the other components of the Postgres system. The installation-wide ODBC configuration file will be placed into the top directory of the Postgres target tree (`POSTGRES DIR`). This can be overridden from the make command-line as

```
% make ODBCINST=filename install
```

### Pre-v6.4 Integrated Installation

If you have a Postgres installation older than v6.4, you have the original source tree available, and you want to use the newest version of the ODBC driver, then you may want to try this form of installation.

1. Copy the output tar file to your target system and unpack it into a clean directory.
2. From the directory containing the sources, type:

```
% ./configure
```

```
% make
```

```
% make POSTGRES DIR=PostgresTopDir install
```

3. If you would like to install components into different trees, then you can specify various destinations explicitly:

```
% make BINDIR=bindir LIBDIR=libdir HEADERDIR=headerdir  
ODBCINST=instfile install
```

### Standalone Installation

A standalone installation is not integrated with or built on the normal Postgres distribution. It should be best suited for building the ODBC driver for multiple, heterogeneous clients who do not have a locally-installed Postgres source tree.

The default location for libraries and headers for the standalone installation is `/usr/local/lib` and `/usr/local/include/iodbc`, respectively. There is another system wide configuration file that gets installed as `/share/odbcinst.ini` (if `/share` exists) or as `/etc/odbcinst.ini` (if `/share` does not exist).

**Note:** Installation of files into `/share` or `/etc` requires system root privileges. Most installation steps for Postgres do not have this requirement, and you can choose another destination which is writable by your non-root Postgres superuser account instead.

1. The standalone installation distribution can be built from the Postgres distribution or may be obtained from Insight Distributors (<http://www.insightdist.com/psqlodbc>), the current maintainers of the non-Unix sources.

Copy the zip or gzipped tarfile to an empty directory. If using the zip package unzip it with the command

```
% unzip -a packagename
```

The `-a` option is necessary to get rid of DOS CR/LF pairs in the source files.

If you have the gzipped tar package than simply run

```
% tar -xzf packagename
```

- a. To create a tar file for a complete standalone installation from the main Postgres source tree:
2. Configure the main Postgres distribution.
  3. Create the tar file:
 

```
% cd interfaces/odbc
% make standalone
```

4. Copy the output tar file to your target system. Be sure to transfer as a binary file if using ftp.
5. Unpack the tar file into a clean directory.
6. Configure the standalone installation:
 

```
% ./configure
```

The configuration can be done with options:

```
% ./configure --prefix=rootdir --with-odbc=inidir
```

where `--prefix` installs the libraries and headers in the directories `rootdir/lib` and `rootdir/include/iodbc`, and `--with-odbc` installs `odbcinst.ini` in the specified directory.

Note that both of these options can also be used from the integrated build but be aware that *when used in the integrated build* `--prefix` will also apply to the rest of your Postgres installation. `--with-odbc` applies only to the configuration file `odbcinst.ini`.

7. Compile and link the source code:
 

```
% make ODBCINST=instdir
```

You can also override the default location for installation on the 'make' command line. This only applies to the installation of the library and header files. Since the driver needs to know the location of the `odbcinst.ini` file attempting to override the environment variable that specifies its installation directory will probably cause you headaches. It is safest simply to allow the driver to install the `odbcinst.ini` file in the default directory or the directory you specified on the './configure' command line with `--with-odbc`.

8. Install the source code:

```
% make POSTGRES DIR=targettree install
```

To override the library and header installation directories separately you need to pass the correct installation variables on the `make install` command line. These variables are `LIBDIR`, `HEADERDIR` and `ODBCINST`. Overriding `POSTGRES DIR` on the make command line will cause `LIBDIR` and `HEADERDIR` to be rooted at the new directory you specify. `ODBCINST` is independent of `POSTGRES DIR`.

Here is how you would specify the various destinations explicitly:

```
% make BINDIR=bindir LIBDIR=libdir HEADERDIR=headerdir install
```

For example, typing

```
% make POSTGRES DIR=/opt/psqlodbc install
```

(after you've used `./configure` and `make`) will cause the libraries and headers to be installed in the directories `/opt/psqlodbc/lib` and `/opt/psqlodbc/include/iodbc` respectively.

The command

```
% make POSTGRES DIR=/opt/psqlodbc HEADERDIR=/usr/local install
```

should cause the libraries to be installed in `/opt/psqlodbc/lib` and the headers in `/usr/local/include/iodbc`. If this doesn't work as expected please contact one of the maintainers.

## Configuration Files

`~/odbc.ini` contains user-specified access information for the `psqlODBC` driver. The file uses conventions typical for Windows Registry files, but despite this restriction can be made to work.

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 = POSTGRES DIR/lib/libpsqlodbc.so
Database=DatabaseName
Servername=localhost
Port=5432
```

**Tip:** Remember that the Postgres database name is usually a single word, without path names of any sort. The Postgres 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
```

# ApplixWare

## Configuration

ApplixWare must be configured correctly in order for it to be able to access the Postgres 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 above. 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 Postgres 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 username 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.

## 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 Postgres ODBC package.

setuid to 256: failed to launch gateway

The September release of ApplixWare v4.4.1 (the first release with official ODBC support under Linux) shows problems when usernames exceed eight (8) characters in length. Problem description ontributed by Steve Campbell (mailto:scampbell@lear.com).

**Author:** Contributed by Steve Campbell (mailto:scampbell@lear.com) on 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 multiple user 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.

## 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 can't find it. That is why `axnet.cnf` needed to be changed.

## 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 Postgres 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.

## Useful Macros

You can add information about your database login and password to the standard Applit startup 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 username and password information.

## Supported Platforms

psqlODBC has been built and tested on Linux. There have been reports of success with FreeBSD and with Solaris. There are no known restrictions on the basic code for other platforms which already support Postgres.

# Chapter 23. JDBC Interface

**Author:** Written by Peter T. Mount (peter@retep.org.uk), the 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.

Postgres provides a *type 4* JDBC Driver. Type 4 indicates that the driver is written in Pure Java, and communicates in the database's own network protocol. Because of this, the driver is platform independent. Once compiled, the driver can be used on any platform.

## Building the JDBC Interface

### Compiling the Driver

The driver's source is located in the `src/interfaces/jdbc` directory of the source tree. To compile simply change directory to that directory, and type:

```
% make
```

Upon completion, you will find the archive `postgresql.jar` in the current directory. This is the JDBC driver.

**Note:** You must use `make`, not `javac`, as the driver uses some dynamic loading techniques for performance reasons, and `javac` cannot cope. The `Makefile` will generate the jar archive.

### Installing the Driver

To use the driver, the jar archive `postgresql.jar` needs to be included in the `CLASSPATH`.

#### Example

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/jdk1.1.6`.

To run the application, I would use:

```
export CLASSPATH =  
/usr/local/lib/finder.jar:/usr/local/lib/postgresql.jar:.  
java uk.org.retep.finder.Main
```

Loading the driver is covered later on in this chapter.

## Preparing the Database for JDBC

Because Java can only use TCP/IP connections, the Postgres postmaster must be running with the `-i` flag.

Also, the `pg_hba.conf` file must be configured. It's located in the PGDATA directory. In a default installation, this file permits access only by Unix domain sockets. For the JDBC driver to connect to the same localhost, you need to add something like:

```
host          all          127.0.0.1      255.255.255.255  password
```

Here access to all databases are possible from the local machine with JDBC.

The JDBC Driver supports trust, ident, password and crypt authentication methods.

## Using the Driver

This section 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.

## Importing JDBC

Any source that uses JDBC needs to import the `java.sql` package, using:

```
import java.sql.*;
```

**Important:** Do not import the `postgresql` package. If you do, your source will not compile, as `javac` will get confused.

## 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 to the best one to use.

In the first method, your code implicitly loads the driver using the `Class.forName()` method. For Postgres, you would use:

```
Class.forName("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`, so you will need to catch it if the driver is not available.

This is the most common method to use, but restricts your code to use just Postgres. If your code may access another database in the future, and you don't use our 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=postgresql.Driver example.ImageView
```

In this example, the JVM will attempt to load the driver as part of its initialisation. Once done, the `ImageView` is started.

Now, this method is the better one to use because it allows your code to be used with other databases, without recompiling the code. The only thing that would also change is the 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 classpath, or the value in the parameter not being correct.

## Connecting to the Database

With JDBC, a database is represented by a URL (Uniform Resource Locator). With Postgres, this takes one of the following forms:

```
jdbc:postgresql:database
```

```
jdbc:postgresql://host/database
```

```
jdbc:postgresql://hostport/database
```

where:

*host*

The hostname of the server. Defaults to "localhost".

*port*

The port number the server is listening on. Defaults to the Postgres 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,user,pwd);
```

## Issuing a Query and Processing the Result

Any time you want to issue SQL statements to the database, you require a `Statement` instance. Once you have a `Statement`, you can use the `executeQuery()` method to issue a query. This will return a `ResultSet` instance, which contains the entire result.

### Using the Statement Interface

The following must be considered when using the `Statement` interface:

You can use a `Statement` instance as many times as you want. You could create one as soon as you open the connection, and use it for the connections lifetime. You have to remember that only one `ResultSet` can exist per `Statement`.

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 the sections covering `Threads` and `Servlets` later in this document if you are thinking of using them, as it covers some important points.

### 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 spec, you should access a field only once. It's safest to stick to this rule, although at the current time, the Postgres 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 with it.

Once you request another query with the `Statement` used to create a `ResultSet`, the currently open instance is closed.

An example is as follows:

```
Statement st = db.createStatement();
ResultSet rs = st.executeQuery("select * from mytable");
while(rs.next()) {
    System.out.print("Column 1 returned ");
    System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

## Performing Updates

To perform an update (or any other SQL statement that does not return a result), you simply use the `executeUpdate()` method:

```
st.executeUpdate("create table basic (a int2, b int2)");
```

## Closing the Connection

To close the database connection, simply call the `close()` method to the `Connection`:

```
db.close();
```

## Using Large Objects

In Postgres, large objects (also known as *blobs*) are used to hold data in the database that cannot be stored in a normal SQL table. They are stored as a Table/Index pair, and are referred to from your own tables by an OID value.

**Important:** For Postgres, you must access large objects within an SQL transaction. Although this has always been true in principle, it was not strictly enforced until the release of v6.5. You would open a transaction by using the `setAutoCommit()` method with an input parameter of `false`:

```
Connection mycon;
...
mycon.setAutoCommit(false);
... now use Large Objects
```

Now, there are two methods of using Large Objects. The first is the standard JDBC way, and is documented here. The other, uses our own extension to the api, which presents the libpq large object API to Java, providing even better access to large objects than the standard. Internally, the driver uses the extension to provide large object support.

In JDBC, the standard way to access them is using the `getBinaryStream()` method in `ResultSet`, and `setBinaryStream()` method in `PreparedStatement`. These methods make the large object appear as a Java stream, allowing you to use the `java.io` package, and others, to manipulate the object.

For example, suppose you have a table containing the file name of an image, and a large object containing that image:

```
create table images (imgname name, imgoid oid);
```

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();
```

Now in this example, `setBinaryStream` transfers a set number of bytes from a stream into a large object, and stores the OID into the field holding a reference to it.

Retrieving an image is even easier (I'm using `PreparedStatement` here, but `Statement` can equally be used):

```
PreparedStatement ps = con.prepareStatement("select oid from images
where name=?");
ps.setString(1,"myimage.gif");
ResultSet rs = ps.executeQuery();
if(rs!=null) {
    while(rs.next()) {
        InputStream is = rs.getBinaryInputStream(1);
        // use the stream in some way here
        is.close();
    }
    rs.close();
}
ps.close();
```

Now here you can see where the Large Object is retrieved as an `InputStream`. You'll also notice that we close the stream before processing the next row in the result. This is part of the JDBC Specification, which states that any `InputStream` returned is closed when `ResultSet.next()` or `ResultSet.close()` is called.

## Postgres Extensions to the JDBC API

Postgres 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.

Now, as these are facilities unique to us, 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.

Accessing the extensions

To access some of the extensions, you need to use some extra methods

in the `postgresql.Connection` class. In this case, you would need to case the return value of `Driver.getConnection()`.

For example:

```
Connection db = Driver.getConnection(url,user,pass);

// later on
Fastpath fp = ((postgresql.Connection)db).getFastpathAPI();
```

Class `postgresql.Connection`

```
java.lang.Object
|
+----postgresql.Connection
```

```
public class Connection extends Object implements Connection
```

These are the extra methods used to gain access to our extensions. I have not listed the methods defined by `java.sql.Connection`.

```
public Fastpath getFastpathAPI() throws SQLException
```

This returns the Fastpath API for the current connection.

NOTE: This is not part of JDBC, but allows access to functions on the postgresql backend itself.

It is primarily used by the LargeObject API

The best way to use this is as follows:

```
import postgresql.fastpath.*;
...
Fastpath fp = ((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 initialising for first time

```
public LargeObjectManager getLargeObjectAPI() throws SQLException
```

This returns the LargeObject API for the current connection.

NOTE: This is not part of JDBC, but allows access to functions on the postgresql backend itself.

The best way to use this is as follows:

```
import postgresql.largeobject.*;
...
```

```

LargeObjectManager lo =
((postgresql.Connection)myconn).getLargeObjectAPI();

```

where myconn is an open Connection to postgresql.

Returns:

LargeObject object that implements the API

Throws: SQLException

by LargeObject when initialising 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.

NOTE: This is not part of JDBC, but an extension.

The best way to use this is as follows:

```

...
((postgresql.Connection)myconn).addDataType("mytype", "my.class.name");
...

```

where myconn is an open Connection to postgresql.

The handling class must extend postgresql.util.PGObject

See Also:

PGObject

### 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's provided because the Large Object API uses it.

To use, you need to import the postgresql.fastpath package, using the line:

```
import postgresql.fastpath.*;
```

Then, in your code, you need to get a FastPath object:

```
Fastpath fp = ((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 postgresql.Connection is required, as the getFastpathAPI() is one of our own methods, not JDBC's.

Once you have a Fastpath instance, you can use the fastpath() methods to execute a backend function.

Class postgresql.fastpath.Fastpath

```
java.lang.Object
|
+----postgresql.fastpath.Fastpath
```

```
public class Fastpath
```

```
extends Object
```

```
This class implements the Fastpath api.
```

This is a means of executing functions imbedded in the postgresql backend from within a java application.

It is based around the file src/interfaces/libpq/fe-exec.c

See Also:

FastpathFastpathArg, LargeObject

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

Throws: SQLException

if a database-access error occurs.

```
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 `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:

byte[] null if no data, Integer if an integer result, or  
 otherwise

Throws: `SQLException`

if name is unknown or if a database-access error occurs.

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 lookup 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.

Parameters:  
name - Function name  
fnid - Function id

```
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.

REMEMBER to close() the resultset after calling this!!

Implementation note about function name lookups:

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 postgresql.LargeObject class performs a query upon it's startup, and passes the returned ResultSet to the addFunctions() method here.

Once this has been done, the LargeObject api refers to the functions by name.

Dont 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.

Parameters:  
rs - ResultSet

Throws: SQLException

if a database-access error occurs.

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.

Parameters:

name - Function name to lookup

Returns:

Function ID for fastpath call

Throws: SQLException

is function is unknown.

Class postgresql.fastpath.FastpathArg

```
java.lang.Object
```

```
|
```

```
+----postgresql.fastpath.FastpathArg
```

```
public class FastpathArg extends Object
```

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 postgresql.largeobject package

See Also:

Fastpath, LargeObjectManager, LargeObject

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.

Parameters:

```
    s - String to store
```

#### Geometric Data Types

PostgreSQL has a set of datatypes that can store geometric features into a table. These range from single points, lines, and polygons.

We support these types in Java with the `postgresql.geometric` package.

It contains classes that extend the `postgresql.util.PGObject` class. Refer to that class for details on how to implement your own data type handlers.

Class `postgresql.geometric.PGbox`

```
java.lang.Object
|
+----postgresql.util.PGObject
|
+----postgresql.geometric.PGbox
```

```
public class PGbox extends PGObject implements Serializable,
Cloneable
```

This represents the box datatype 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 postgresql.geometric.PGcircle

```
java.lang.Object
|
+----postgresql.util.PGObject
|
+----postgresql.geometric.PGcircle
```

```
public class PGcircle extends PGObject implements Serializable,
Cloneable
```

This represents postgresql's circle datatype, consisting of a point and a radius

Variables

```
public PGpoint center

        This is the centre point

public double radius

        This is the radius
```

Constructors

```
public PGcircle(double x,
                double y,
                double r)
```

Parameters:  
     x - coordinate of centre  
     y - coordinate of centre  
     r - radius of circle

```
public PGcircle(PGpoint c,
                double r)
```

Parameters:  
     c - PGpoint describing the circle's centre  
     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 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 PGcircle in the syntax expected by postgresql

Overrides:  
getValue in class PGobject

Class postgresql.geometric.PGline

java.lang.Object

|

+----postgresql.util.PGobject

|

+----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 circle 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 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 PGLine in the syntax expected by postgresql

    Overrides:
        getValue in class PGObject

```

Class postgresql.geometric.PGLseg

```

java.lang.Object
|
+----postgresql.util.PGObject
|
+----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 circle 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 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 PGlseg in the syntax expected by postgresql

Overrides:  
     getValue in class PGObject

Class postgresql.geometric.PGpath

```
java.lang.Object
|
+----postgresql.util.PGObject
|
+----postgresql.geometric.PGpath
```

public class PGpath extends PGObject implements Serializable, Cloneable

This implements a path (a multiple 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 circle 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 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()

    This returns the polygon 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 postgresql.geometric.PGpoint

java.lang.Object
|
+----postgresql.util.PGObject
    |
    +----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` datatype 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 imbedded 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 boxes are identical

Overrides:  
     equals in class PObject

public Object clone()

    This must be overridden to allow the object to be cloned

Overrides:  
     clone in class PObject

public String getValue()

Returns:  
     the PGpoint in the syntax expected by postgresql

Overrides:  
     getValue in class PObject

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 postgresql.geometric.PGpolygon

```
java.lang.Object
```

```
|
```

```
+----+postgresql.util.PGobject
```

```
|
```

```
+----+postgresql.geometric.PGpolygon
```

```
public class PGpolygon extends PGobject implements Serializable,
Cloneable
```

This implements the polygon datatype 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 circle 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 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 PGpolygon in the syntax expected by postgresql

Overrides:  
 getValue in class PGObject

#### 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 `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.

Class `postgresql.largeobject.LargeObject`

```
java.lang.Object
|
+----postgresql.largeobject.LargeObject
```

```
public class LargeObject extends Object
```

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 `getAsciiStream`, `getBinaryStream`, or `getUnicodeStream` methods in `ResultSet`, or `setAsciiStream`, `setBinaryStream`, or `setUnicodeStream` methods in `PreparedStatement` to access Large Objects.

However, sometimes lower level access to Large Objects are required, that are not supported by the JDBC specification.

Refer to `postgresql.largeobject.LargeObjectManager` on how to gain access to a Large Object, or how to create one.

See Also:  
     `LargeObjectManager`

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
```

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.

Throws: `SQLException`  
if a database-access error occurs.

```
public byte[] read(int len) throws SQLException
```

Reads some data from the object, and return as a `byte[]` array

Parameters:  
len - number of bytes to read

Returns:  
`byte[]` array containing data read

Throws: `SQLException`  
if a database-access error occurs.

```
public void 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

Throws: `SQLException`  
if a database-access error occurs.

```
public void write(byte buf[]) throws SQLException
```

Writes an array to the object

Parameters:  
buf - array to write

Throws: `SQLException`  
if a database-access error occurs.

```
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

Throws: SQLException  
 if a database-access error occurs.

```
public void seek(int pos,
                int ref) throws SQLException
```

Sets the current position within the object.

This is similar to the fseek() call in the standard C library. It allows you to have random access to the large object.

Parameters:

pos - position within object  
 ref - Either SEEK\_SET, SEEK\_CUR or SEEK\_END

Throws: SQLException  
 if a database-access error occurs.

```
public void seek(int pos) throws SQLException
```

Sets the current position within the object.

This is similar to the fseek() call in the standard C library. It allows you to have random access to the large object.

Parameters:

pos - position within object from beginning

Throws: SQLException  
 if a database-access error occurs.

```
public int tell() throws SQLException
```

Returns:

the current position within the object

Throws: SQLException  
 if a database-access error occurs.

```
public int size() throws SQLException
```

This method is inefficient, as the only way to find out the size of the object is to seek to the end, record the current position, then return to the original position.

A better method will be found in the future.

Returns:

the size of the large object

Throws: SQLException  
 if a database-access error occurs.

```
public InputStream getInputStream() throws SQLException
```

Returns an InputStream from this object.

This InputStream can then be used in any method that requires an InputStream.

Throws: SQLException  
if a database-access error occurs.

```
public OutputStream getOutputStream() throws SQLException
```

Returns an OutputStream to this object

This OutputStream can then be used in any method that requires an OutputStream.

Throws: SQLException  
if a database-access error occurs.

Class postgresql.largeobject.LargeObjectManager

```
java.lang.Object
|
+----postgresql.largeobject.LargeObjectManager
```

```
public class LargeObjectManager extends Object
```

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 postgresql.largeobject.LargeObject is returned, and its methods then allow access to the object.

This class can only be created by postgresql.Connection

To get access to this class, use the following segment of code:

```
import postgresql.largeobject.*;
Connection conn;
LargeObjectManager lobj;
... code that opens a connection ...
lobj = ((postgresql.Connection)myconn).getLargeObjectAPI();
```

Normally, client code would use the getAsciiStream, getBinaryStream, or getUnicodeStream methods in ResultSet, or setAsciiStream, setBinaryStream, or setUnicodeStream methods in PreparedStatement to access Large Objects.

However, sometimes lower level access to Large Objects are required, that are not supported by the JDBC specification.

Refer to postgresql.largeobject.LargeObject on how to manipulate the contents of a Large Object.

See Also:

## LargeObject

## 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

## 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).

## Parameters:

oid - of large object

## Returns:

LargeObject instance providing access to the object

## Throws: SQLException

on error

```
public LargeObject open(int oid,
                        int mode) throws SQLException
```

This opens an existing large object, based on its OID

## Parameters:

oid - of large object

mode - mode of open

## Returns:

LargeObject instance providing access to the object

## Throws: SQLException

on error

```
public int create() throws SQLException
```

This creates a large object, returning its OID.

It defaults to READWRITE for the new object's attributes.

## Returns:

oid of new object

Throws: SQLException  
on error

```
public int create(int mode) throws SQLException
```

This creates a large object, returning its OID

Parameters:  
mode - a bitmask describing different attributes of  
the  
new object

Returns:  
oid of new object

Throws: SQLException  
on error

```
public void delete(int oid) throws SQLException
```

This deletes a large object.

Parameters:  
oid - describing object to delete

Throws: SQLException  
on error

```
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.

Parameters:  
oid - describing object to delete

Throws: SQLException  
on error

#### Object Serialisation

PostgreSQL is not a normal SQL Database. It is far more extensible than most other databases, and does support Object Oriented features that are unique to it.

One of the consequences of this, is that you can have one table refer to a row in another table. For example:

```
test=> create table users (username name,fullname text);
CREATE
test=> create table server (servername name,adminuser users);
CREATE
```

```

test=> insert into users values ('peter','Peter Mount');
INSERT 2610132 1
test=> insert into server values ('maidast',2610132::users);
INSERT 2610133 1
test=> select * from users;
username|fullname
-----+-----
peter   |Peter Mount
(1 row)

test=> select * from server;
servername|adminuser
-----+-----
maidast   | 2610132
(1 row)

```

Ok, the above example shows that we can use a table name as a field, and the row's oid value is stored in that field.

What does this have to do with Java?

In Java, you can store an object to a Stream as long as it's class implements the `java.io.Serializable` interface. This process, known as Object Serialization, can be used to store complex objects into the database.

Now, under JDBC, you would have to use a `LargeObject` to store them. However, you cannot perform queries on those objects.

What the `postgresql.util.Serialize` class does, is provide a means of storing an object as a table, and to retrieve that object from a table. In most cases, you would not need to access this class direct, but you would use the `PreparedStatement.setObject()` and `ResultSet.getObject()` methods. Those methods will check the objects class name against the table's in the database. If a match is found, it assumes that the object is a Serialized object, and retrieves it from that table. As it does so, if the object contains other serialized objects, then it recurses down the tree.

Sound's complicated? In fact, it's simpler than what I wrote - it's just difficult to explain.

The only time you would access this class, is to use the `create()` methods. These are not used by the driver, but issue one or more "create table" statements to the database, based on a Java Object or Class that you want to serialize.

Oh, one last thing. If your object contains a line like:

```
public int oid;
```

then, when the object is retrieved from the table, it is set to the oid within the table. Then, if the object is modified, and re-serialized, the existing entry is updated.

If the oid variable is not present, then when the object is serialized, it is always inserted into the table, and any existing entry in the table is preserved.

Setting oid to 0 before serialization, will also cause the object to be inserted. This enables an object to be duplicated in the database.

Class `postgresql.util.Serialize`

```
java.lang.Object
|
+----postgresql.util.Serialize

public class Serialize extends Object
```

This class uses PostgreSQL's object oriented features to store Java Objects. It does this by mapping a Java Class name to a table in the database. Each entry in this new table then represents a Serialized instance of this class. As each entry has an OID (Object Identifier), this OID can be included in another table. This is too complex to show here, and will be documented in the main documents in more detail.

Constructors

```
public Serialize(Connection c,
                  String type) throws SQLException
```

This creates an instance that can be used to serialize or deserialize a Java object from a PostgreSQL table.

Methods

```
public Object fetch(int oid) throws SQLException
```

This fetches an object from a table, given it's OID

Parameters:

oid - The oid of the object

Returns:

Object relating to oid

Throws: SQLException

on error

```
public int store(Object o) throws SQLException
```

This stores an object into a table, returning it's OID.

If the object has an int called OID, and it is > 0, then that value is used for the OID, and the table will be updated. If the value of OID is 0, then a new row will be created, and the value of

OID will be set in the object. This enables an object's value in the database to be updateable. If the object has no int called OID, then the object is stored. However if the object is later retrieved, amended and stored again, it's new state will be appended to the table, and will not overwrite the old entries.

Parameters:

o - Object to store (must implement Serializable)

Returns:

oid of stored object

Throws: SQLException

on error

```
public static void create(Connection con,
                        Object o) throws SQLException
```

This method is not used by the driver, but it creates a table, given a Serializable Java Object. It should be used before serializing any objects.

Parameters:

c - Connection to database

o - Object to base table on

Throws: SQLException

on error

Returns:

Object relating to oid

Throws: SQLException

on error

```
public int store(Object o) throws SQLException
```

This stores an object into a table, returning it's OID.

If the object has an int called OID, and it is > 0, then that value is used for the OID, and the table will be updated. If the value of OID is 0, then a new row will be created, and the value of OID will be set in the object. This enables an object's value in the database to be updateable. If the object has no int called OID, then the object is stored. However if the object is later retrieved, amended and stored again, it's new state will be appended to the table, and will not overwrite the old entries.

Parameters:

o - Object to store (must implement Serializable)

Returns:

oid of stored object

Throws: SQLException

on error

```
public static void create(Connection con,
                        Object o) throws SQLException
```

This method is not used by the driver, but it creates a table, given a Serializable Java Object. It should be used before serializing any objects.

Parameters:

c - Connection to database  
o - Object to base table on

Throws: SQLException  
on error

```
public static void create(Connection con,
                        Class c) throws SQLException
```

This method is not used by the driver, but it creates a table, given a Serializable Java Object. It should be used before serializing any objects.

Parameters:

c - Connection to database  
o - Class to base table on

Throws: SQLException  
on error

```
public static String toPostgreSQL(String name) throws SQLException
```

This converts a Java Class name to a postgresql table, by replacing . with \_

Because of this, a Class name may not have \_ in the name.

Another limitation, is that the entire class name (including packages) cannot be longer than 31 characters (a limit forced by PostgreSQL).

Parameters:

name - Class name

Returns:

PostgreSQL table name

Throws: SQLException  
on error

```
public static String toClassName(String name) throws SQLException
```

This converts a postgresql table to a Java Class name, by replacing \_ with .

Parameters:

name - PostgreSQL table name

Returns:  
     Class name

Throws: SQLException  
         on error

#### Utility Classes

The postgresql.util package contains classes used by the internals of the main driver, and the other extensions.

Class postgresql.util.PGmoney

```
java.lang.Object
|
+----postgresql.util.PGobject
|
+----postgresql.util.PGmoney
```

```
public class PGmoney extends PGobject implements Serializable,
Cloneable
```

This implements a class that handles the PostgreSQL money type

#### Variables

```
public double val

    The value of the field
```

#### Constructors

```
public PGmoney(double value)
```

Parameters:  
     value - of field

```
public PGmoney(String value) throws SQLException
```

This is called mainly from the other geometric types, when a point is imbedded within their definition.

Parameters:  
     value - Definition of this point in PostgreSQL's syntax

```
public PGmoney()
```

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 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 PGpoint in the syntax expected by postgresql

Overrides:
    getValue in class PGObject

```

Class postgresql.util.PGObject

```

java.lang.Object
|
+----postgresql.util.PGObject

```

```

public class PGObject extends Object implements Serializable,
Cloneable

```

This class is used to describe data types that are unknown by JDBC Standard.

A call to `postgresql.Connection` permits a class that extends this class to be associated with a named type. This is how the `postgresql.geometric` package operates.

`ResultSet.getObject()` will return this class for any type that is not recognised on having it's own handler. Because of this, any postgresql data type is supported.

Constructors

```
public PGObject()
```

This is called by `postgresql.Connection.getObject()` to create the object.

Methods

```
public final void setType(String type)
```

This method sets the type of this object.

It should not be extended by subclasses, hence its final

Parameters:

type - a string describing the type of the object

```
public void setValue(String value) throws SQLException
```

This method sets the value of this object. It must be overridden.

Parameters:

value - a string representation of the value of the object

Throws: `SQLException`

thrown if value is invalid for this type

```
public final String getType()
```

As this cannot change during the life of the object, it's final.

Returns:

the type name of this object

```
public String getValue()
```

This must be overridden, to return the value of the object, in the form required by `postgresql`.

Returns:

the value of this object

```
public boolean equals(Object obj)
```

This must be overridden to allow comparisons of objects

Parameters:

obj - Object to compare with

Returns:

true if the two boxes are identical

Overrides:

`equals` in class `Object`

```
public Object clone()
```

This must be overridden to allow the object to be cloned

Overrides:

clone in class Object

```
public String toString()
```

This is defined here, so user code need not override it.

Returns:

the value of this object, in the syntax expected by  
postgresql

Overrides:

toString in class Object

Class postgresql.util.PGtokenizer

```
java.lang.Object
```

```
|
```

```
+----postgresql.util.PGtokenizer
```

```
public class PGtokenizer extends Object
```

This class is used to tokenize the text output of postgres.

We could have used StringTokenizer to do this, however, we needed to handle nesting of '(' ')' '[' ']' '<' and '>' as these are used by the geometric data types.

It's mainly used by the geometric classes, but is useful in parsing any output from custom data types output from postgresql.

See Also:

PGbox, PGcircle, PGlseg, PGpath, PGpoint, PGpolygon

Constructors

```
public PGtokenizer(String string,
                  char delim)
```

Create a tokeniser.

Parameters:

string - containing tokens

delim - single character to split the tokens

Methods

```
public int tokenize(String string,
                  char delim)
```

This resets this tokenizer with a new string and/or delimiter.

Parameters:

string - containing tokens  
 delim - single character to split the tokens

```
public int getSize()
```

Returns:

the number of tokens available

```
public String getToken(int n)
```

Parameters:

n - Token number ( 0 ... getSize()-1 )

Returns:

The token value

```
public PGtokenizer tokenizeToken(int n,
                                char delim)
```

This returns a new tokenizer based on one of our tokens. The geometric datatypes use this to process nested tokens (usually PGpoint).

Parameters:

n - Token number ( 0 ... getSize()-1 )  
 delim - The delimiter to use

Returns:

A new instance of PGtokenizer based on the token

```
public static String remove(String s,
                            String l,
                            String t)
```

This removes the lead/trailing strings from a string

Parameters:

s - Source string  
 l - Leading string to remove  
 t - Trailing string to remove

Returns:

String without the lead/trailing strings

```
public void remove(String l,
                  String t)
```

This removes the lead/trailing strings from all tokens

Parameters:

l - Leading string to remove  
 t - Trailing string to remove

```

public static String removePara(String s)

    Removes ( and ) from the beginning and end of a string

Parameters:
    s - String to remove from

Returns:
    String without the ( or )

public void removePara()

    Removes ( and ) from the beginning and end of all tokens

Returns:
    String without the ( or )

public static String removeBox(String s)

    Removes [ and ] from the beginning and end of a string

Parameters:
    s - String to remove from

Returns:
    String without the [ or ]

public void removeBox()

    Removes [ and ] from the beginning and end of all tokens

Returns:
    String without the [ or ]

public static String removeAngle(String s)

    Removes < and > from the beginning and end of a string

Parameters:
    s - String to remove from

Returns:
    String without the < or >

public void removeAngle()

    Removes < and > from the beginning and end of all tokens

Returns:
    String without the < or >

```

Class `postgresql.util.Serialize`

This was documented earlier under Object Serialisation.

Class `postgresql.util.UnixCrypt`



Check that enteredPassword encrypts to encryptedPassword.

Parameters:

encryptedPassword - The encryptedPassword. The first two characters are assumed to be the salt. This string would be the same as one found in a Unix /etc/passwd file.

enteredPassword - The password as entered by the user (or otherwise aquired).

Returns:

true if the password should be considered correct.

Using the driver in a multi Threaded or 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.

PostgreSQL 6.4, brings thread safety to the entire driver. Standard JDBC was thread safe in 6.3.x, but the Fastpath API wasn't.

So, if your application uses multiple threads (which most decent ones would), then you don't have to worry about complex schemes to ensure only one uses the database at any time.

If a thread attempts to use the connection while another is using it, it will wait until the other thread has finished it's current operation.

If it's a standard SQL statement, then the operation is sending the statement, and retrieving any ResultSet (in full).

If it's a Fastpath call (ie: reading a block from a LargeObject), then it's the time to send, and retrieve that block.

This is fine for applications & 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 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. It 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's dead.

So, with servlets, it's 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 is created for each Connection.

It's up to you, and your applications requirements.

## 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 on JavaSoft's web site (<http://www.javasoft.com>).

My own web site (<http://www.retep.org.uk>) contains updated information not included in this document, and also includes precompiled drivers for v6.4, and earlier.

## Chapter 24. Lisp Programming Interface

`pg.el` is a socket-level interface to Postgres for emacs.

**Author:** Written by Eric Marsden (mailto:emarsden@mail.dotcom.fr) on 21 Jul 1999.

`pg.el` is a socket-level interface to Postgres for emacs (text editor extraordinaire). The module is capable of type coercions from a range of SQL types to the equivalent Emacs Lisp type. It currently supports neither crypt or Kerberos authentication, nor large objects.

The code (version 0.2) is available under GNU GPL from Eric Marsden (<http://www.chez.com/emarsden/downloads/pg.el>)

Changes since last release:

- now works with XEmacs (tested with Emacs 19.34 & 20.2, and XEmacs 20.4)
- added functions to provide database metainformation (list of databases, of tables, of columns)
- arguments to 'pg:result' are now :keywords
- MULE-resistant
- more self-testing code

Please note that this is a programmer's API, and doesn't provide any form of user interface. Example:

```
(defun demo ()
  (interactive)
  (let* ((conn (pg:connect "template1" "postgres" "postgres"))
        (res (pg:exec conn "SELECT * from scshdemo WHERE a =
42"))))
    (message "status is %s" (pg:result res :status))
    (message "metadata is %s" (pg:result res :attributes))
    (message "data is %s" (pg:result res :tuples))
    (pg:disconnect conn))
```

# Chapter 25. Postgres Source Code

## 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 Postgres 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 26. Overview of PostgreSQL Internals

**Author:** This chapter originally appeared as a part of *Simkovics, 1998*, 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 Postgres. 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 Postgres 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.

## 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 Postgres 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 Postgres's internal control and data structures.

## How Connections are Established

Postgres 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 `libpq` library. Note that applications implemented using `ecpg` (the Postgres 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.

## 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.

## 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 Postgres 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 Postgres for the processing of a query we use an example to illustrate the changes made to these data structures in every stage.

### Example 26-1. A Simple Select

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.

```
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 *A Simple Select*. 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. `select s.sname, se.pno from supplier s, sells se where s.sno > 2 and s.sno = se.sno;` (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 *A Simple Select*. 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. `select s.sname, se.pno from supplier s, sells se where s.sno > 2 and s.sno = se.sno;` 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.

## 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 which 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* which 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.

## The Postgres Rule System

Postgres supports a powerful *rule system* for the specification of *views* and ambiguous *view updates*. Originally the Postgres 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 Postgres 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 Postgres system refer to *The PostgreSQL User's Guide*.

## 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.

## 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 *Stonebraker et al, 1989*).

### **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';
```

## 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.

### Generating Possible Plans

The planner/optimizer decides which plans should be generated based upon the types of indices 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 indices 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.

### 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 which 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, Postgres 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.

## 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 27. pg\_options

**Note:** Contributed by Massimo Dal Zotto (mailto:dz@cs.unitn.it)

The optional file `data/pg_options` contains runtime options used by the backend to control trace messages and other backend tunable parameters. What makes this file interesting is the fact that it is re-read by a backend when it receives a SIGHUP signal, making thus possible to change run-time options on the fly without needing to restart Postgres. The options specified in this file may be debugging flags used by the trace package (`backend/utls/misc/trace.c`) or numeric parameters which can be used by the backend to control its behaviour. New options and parameters must be defined in `backend/utls/misc/trace.c` and `backend/include/utls/trace.h`.

For example suppose we want to add conditional trace messages and a tunable numeric parameter to the code in file `foo.c`. All we need to do is to add the constant `TRACE_FOO` and `OPT_FOO_PARAM` into `backend/include/utls/trace.h`:

```
/* file trace.h */
enum pg_option_enum {
    ...
    TRACE_FOO,                /* trace foo functions */
    OPT_FOO_PARAM,           /* foo tunable parameter */

    NUM_PG_OPTIONS           /* must be the last item of enum */
};
```

and a corresponding line in `backend/utls/misc/trace.c`:

```
/* file trace.c */
static char *opt_names[] = {
    ...
    "foo",                    /* trace foo functions */
    "fooparam"               /* foo tunable parameter */
};
```

Options in the two files must be specified in exactly the same order. In the `foo` source files we can now reference the new flags with:

```
/* file foo.c */
#include "trace.h"
#define foo_param pg_options[OPT_FOO_PARAM]

int
foo_function(int x, int y)
{
    TPRINTF(TRACE_FOO, "entering foo_function, foo_param=%d",
foo_param);
    if (foo_param > 10) {
        do_more_foo(x, y);
    }
}
```

```

    }
}

```

Existing files using private trace flags can be changed by simply adding the following code:

```

#include "trace.h"
/* int my_own_flag = 0; -- removed */
#define my_own_flag pg_options[OPT_MY_OWN_FLAG]

```

All `pg_options` are initialized to zero at backend startup. If we need a different default value we must add some initialization code at the beginning of `PostgresMain`. Now we can set the `foo_param` and enable `foo` trace by writing values into the `data/pg_options` file:

```

# file pg_options
...
foo=1
fooparam=17

```

The new options will be read by all new backends when they are started. To make effective the changes for all running backends we need to send a `SIGHUP` to the postmaster. The signal will be automatically sent to all the backends. We can also activate the changes only for a specific backend by sending the `SIGHUP` directly to it.

`pg_options` can also be specified with the `-T` switch of `Postgres`:

```

postgres options -T "verbose=2,query,hostlookup-"

```

The functions used for printing errors and debug messages can now make use of the `syslog(2)` facility. Message printed to `stdout` or `stderr` are prefixed by a timestamp containing also the backend pid:

```

#timestamp          #pid    #message
980127.17:52:14.173 [29271] StartTransactionCommand
980127.17:52:14.174 [29271] ProcessUtility: drop table t;
980127.17:52:14.186 [29271] SIIncNumEntries: table is 70% full
980127.17:52:14.186 [29286] Async_NotifyHandler
980127.17:52:14.186 [29286] Waking up sleeping backend process
980127.19:52:14.292 [29286] Async_NotifyFrontEnd
980127.19:52:14.413 [29286] Async_NotifyFrontEnd done
980127.19:52:14.466 [29286] Async_NotifyHandler done

```

This format improves readability of the logs and allows people to understand exactly which backend is doing what and at which time. It also makes easier to write simple awk or perl scripts which monitor the log to detect database errors or problem, or to compute transaction time statistics.

Messages printed to syslog use the log facility LOG\_LOCAL0. The use of syslog can be controlled with the `syslog pg_option`. Unfortunately many functions call directly `printf()` to print their messages to `stdout` or `stderr` and this output can't be redirected to syslog or have timestamps in it. It would be advisable that all calls to `printf` would be replaced with the `PRINTF` macro and output to `stderr` be changed to use `EPRINTF` instead so that we can control all output in a uniform way.

The new `pg_options` mechanism is more convenient than defining new backend option switches because:

- we don't have to define a different switch for each thing we want to control. All options are defined as keywords in an external file stored in the data directory.

- we don't have to restart Postgres to change the setting of some option. Normally backend options are specified to the postmaster and passed to each backend when it is started. Now they are read from a file.

- we can change options on the fly while a backend is running. We can thus investigate some problem by activating debug messages only when the problem appears. We can also try different values for tunable parameters.

The format of the `pg_options` file is as follows:

```
# comment
option=integer_value # set value for option
option                # set option = 1
option+               # set option = 1
option-               # set option = 0
```

Note that *keyword* can also be an abbreviation of the option name defined in `backend/utils/misc/trace.c`.

Refer to *The Administrator's Guide* chapter on runtime options for a complete list of currently supported options.

Some of the existing code using private variables and option switches has been changed to make use of the `pg_options` feature, mainly in `postgres.c`. It would be advisable to modify all existing code in this way, so that we can get rid of many of the switches on the Postgres command line and can have more tunable options with a unique place to put option values.

# Chapter 28. Genetic Query Optimization in Database Systems

**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.

## 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, index scan, merge join in Postgres) to process individual **joins** and a diversity of *indices* (e.g., r-tree, b-tree, hash in Postgres) as access paths for relations.

The current Postgres 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 Postgres 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 within exploring the space of possible query plans arose 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.

## Genetic Algorithms (GA)

The 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 adaption 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).



The GEQO module gives the following benefits to the Postgres DBMS compared to the Postgres query optimizer implementation:

- Handling of large **join** queries through non-exhaustive search;
- Improved cost size approximation of query plans since no longer plan merging is needed (the GEQO module evaluates the cost for a query plan as an individual).

## Future Implementation Tasks for Postgres GEQO

### Basic Improvements

#### Improve freeing of memory when query is already processed

With large **join** queries the computing time spent for the genetic query optimization seems to be a mere *fraction* of the time Postgres needs for freeing memory via routine `MemoryContextFree`, file `backend/utils/mmgr/mcxt.c`. Debugging showed that it get stucked in a loop of routine `OrderedElemPop`, file `backend/utils/mmgr/oset.c`. The same problems arise with long queries when using the normal Postgres query optimization algorithm.

#### Improve 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

#### Find better solution for integer overflow

In file `backend/optimizer/geqo/geqo_eval.c`, routine `geqo_joinrel_size`, the present hack for MAXINT overflow is to set the Postgres integer value of `rel->size` to its logarithm. Modifications of `Rel` in `backend/nodes/relation.h` will surely have severe impacts on the whole Postgres implementation.

#### Find solution for exhausted memory

Memory exhaustion may occur with more than 10 relations involved in a query. In file `backend/optimizer/geqo/geqo_eval.c`, routine `gimme_tree` is recursively called. Maybe I forgot something to be freed correctly, but I dunno what. Of course the `rel` data structure of the **join** keeps growing and growing the more relations are packed into it. Suggestions are welcome :-)

## References

Reference information for GEQ algorithms.

*The Hitch-Hiker's Guide to Evolutionary Computation*, Jörg Heitkötter and David Beasley, InterNet resource, *The Design and Implementation of the Postgres Query Optimizer*, Z. Fong, University of California, Berkeley Computer Science Department, *Fundamentals of Database Systems*, R. Elmasri and S. Navathe, The Benjamin/Cummings Pub., Inc..

FAQ in comp.ai.genetic (news://comp.ai.genetic) is available at Encore  
(ftp://ftp.Germany.EU.net/pub/research/softcomp/EC/Welcome.html).

File `planner/Report.ps` in the 'postgres-papers' distribution.

# Chapter 29. Frontend/Backend Protocol

**Note:** Written by Phil Thompson (mailto:phil@river-bank.demon.co.uk). Updates for protocol 2.0 by Tom Lane (mailto:tgl@sss.pgh.pa.us).

Postgres uses a message-based protocol for communication between frontends and backends. The protocol is implemented over TCP/IP and also on Unix sockets. Postgres v6.3 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 Postgres v6.4 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.

## Overview

The three major components are the frontend (running on the client) and the postmaster and backend (running on the server). The postmaster and backend have different roles but may be implemented by the same executable.

A frontend sends a startup packet to the postmaster. This includes the names of the user and the database the user wants to connect to. The postmaster then uses this, and the information in the `pg_hba.conf(5)` 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 postmaster validates this it responds to the frontend that it is authenticated and hands over the connection to a backend. The backend then sends a message indicating successful startup (normal case) or failure (for example, an invalid database name).

Subsequent communications are query and result packets exchanged between the frontend and the backend. The postmaster takes no further part in ordinary query/result communication. (However, the postmaster is involved when the frontend wishes to cancel a query currently being executed by its backend. Further details about that appear below.)

When the frontend wishes to disconnect it sends an appropriate packet and closes the connection without waiting for a response for the backend.

Packets are sent as a data stream. The first byte determines what should be expected in the rest of the packet. The exception is packets sent from a frontend to the postmaster, which comprise a packet length then the packet itself. The difference is historical.

## Protocol

This section describes the message flow. There are four different types of flows depending on the state of the connection: startup, query, function call, and termination. There are also special provisions for notification responses and command cancellation, which can occur at any time after the startup phase.

## Startup

Startup is divided into an authentication phase and a backend startup phase.

Initially, the frontend sends a `StartupPacket`. The postmaster uses this info and the contents of the `pg_hba.conf(5)` file to determine what authentication method the frontend must use. The postmaster then responds with one of the following messages:

### ErrorResponse

The postmaster then immediately closes the connection.

### AuthenticationOk

The postmaster then hands over to the backend. The postmaster takes no further part in the communication.

### AuthenticationKerberosV4

The frontend must then take part in a Kerberos V4 authentication dialog (not described here) with the postmaster. If this is successful, the postmaster 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) with the postmaster. If this is successful, the postmaster responds with an `AuthenticationOk`, otherwise it responds with an `ErrorResponse`.

### AuthenticationUnencryptedPassword

The frontend must then send an `UnencryptedPasswordPacket`. If this is the correct password, the postmaster responds with an `AuthenticationOk`, otherwise it responds with an `ErrorResponse`.

### AuthenticationEncryptedPassword

The frontend must then send an `EncryptedPasswordPacket`. If this is the correct password, the postmaster responds with an `AuthenticationOk`, otherwise it responds with an `ErrorResponse`.

If the frontend does not support the authentication method requested by the postmaster, then it should immediately close the connection.

After sending `AuthenticationOk`, the postmaster attempts to launch a backend process. Since this might fail, or the backend might encounter a failure during startup, the frontend must wait for the backend to acknowledge successful startup. The frontend should send no messages at this point. The possible messages from the backend during this phase are:

### BackendKeyData

This message is issued after successful backend startup. It 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**

Backend startup is successful. The frontend may now issue query or function call messages.

**ErrorResponse**

Backend startup 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 startup phase), or to consider ReadyForQuery as ending the startup phase and each subsequent query cycle.

## 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 relation. 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 relation to the frontend. It then sends a CopyDataRows message, and then a CompletedResponse message with a tag of "COPY".

**CursorResponse**

The query was either an insert(l), delete(l), update(l), fetch(l) or a select(l) command. If the transaction has been aborted then the backend sends a CompletedResponse message with a tag of "\*ABORT STATE\*". Otherwise the following responses are sent.

For an insert(l) command, the backend then sends a CompletedResponse message with a tag of "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(l) command, the backend then sends a CompletedResponse message with a tag of "DELETE *rows*" where *rows* is the number of rows deleted.

For an `update(l)` command, the backend then sends a `CompletedResponse` message with a tag of `"UPDATE rows"` where `rows` is the number of rows deleted.

For a `fetch(l)` or `select(l)` command, the backend sends a `RowDescription` message. This is then followed by an `AsciiRow` or `BinaryRow` message (depending on whether a binary cursor was specified) for each row being returned to the frontend. Finally, the backend sends a `CompletedResponse` message with a tag of `"SELECT"`.

#### EmptyQueryResponse

An empty query string was recognized. (The need to specially distinguish this case is historical.)

#### 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, ie. 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.

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 postmaster. 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(l)` commands then it must be prepared to accept `NotificationResponse` messages at any time; see below.

## 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, ie. 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(l)` commands then it must be prepared to accept `NotificationResponse` messages at any time; see below.

## Notification Responses

If a frontend issues a `listen(l)` command, then the backend will send a `NotificationResponse` message (not to be confused with `NoticeResponse`!) whenever a `notify(l)` command is executed for the same notification name.

Notification responses are permitted at any point in the protocol (after startup), 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(l)` command has been executed for a name for which a previous `listen(l)` 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.

## Cancelling Requests in Progress

During the processing of a query, the frontend may request cancellation of the query by sending an appropriate request to the postmaster. The cancel request is not sent directly 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 postmaster and sends a `CancelRequest` message, rather than the `StartupPacket` message that would ordinarily be sent across a new connection. The postmaster 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 startup. If the request matches the PID and secret key for a currently executing backend, the postmaster signals the backend to abort processing of the current 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 to the postmaster 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.

## 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 postmaster, if it doesn't want to terminate itself.

## 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 `'\0'` terminated string. The `'\0'` 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 `'\0'` 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').

## Message Formats

This section describes the detailed format of each message. Each can be sent by either a frontend (F), a postmaster/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 '\0' 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.

AuthenticationUnencryptedPassword (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(3)

Specifies that an unencrypted password is required.

AuthenticationEncryptedPassword (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(4)

Specifies that an encrypted password is required.

Byte2

The salt to use when encrypting the password.

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.)

Byten

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.

Byten

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 (but not always) a single word that identifies which SQL command was completed.

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.

EncryptedPasswordPacket (F)

Int32

The size of the packet in bytes.

String

The encrypted (using crypt()) password.

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.

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.

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 by the postmaster.

LimString64

Unused.

LimString64

The optional tty the backend should use for debugging messages.

Terminate (F)

Byte1('X')

Identifies the message as a termination.

UnencryptedPasswordPacket (F)

Int32

The size of the packet in bytes.

String

The unencrypted password.

# Chapter 30. Postgres Signals

**Note:** Contributed by Massimo Dal Zotto (mailto:dz@cs.unitn.it)

Postgres uses the following signals for communication between the postmaster and backends:

**Table 30-1. Postgres Signals**

Signal	postmaster Action	Server Action
SIGHUP	kill(*,sighup)	read_pg_options
SIGINT	die	cancel query
SIGQUIT	kill(*,sigterm)	handle_warn
SIGTERM	kill(*,sigterm), kill(*,9), die	die
SIGPIPE	ignored	die
SIGUSR1	kill(*,sigusr1), die	quickdie
SIGUSR2	kill(*,sigusr2)	async notify (SI flush)
SIGCHLD	reaper	ignored (alive test)
SIGTTIN	ignored	
SIGTTOU	ignored	
SIGCONT	dumpstatus	
SIGFPE		FloatExceptionHandler

**Note:** kill(\*,signal) means sending a signal to all backends.

The main changes to the old signal handling are the use of SIGQUIT instead of SIGHUP to handle warns, SIGHUP to re-read the pg\_options file and the redirection to all active backends of SIGHUP, SIGTERM, SIGUSR1 and SIGUSR2 sent to the postmaster. In this way these signals sent to the postmaster can be sent automatically to all the backends without need to know their pids. To shut down postgres one needs only to send a SIGTERM to postmaster and it will stop automatically all the backends.

The SIGUSR2 signal is also used to prevent SI cache table overflow which happens when some backend doesn't process SI cache for a long period. When a backend detects the SI table full at 70% it simply sends a signal to the postmaster which will wake up all idle backends and make them flush the cache.

The typical use of signals by programmers could be the following:

```
# stop postgres
kill -TERM $postmaster_pid

# kill all the backends
kill -QUIT $postmaster_pid

# kill only the postmaster
kill -INT $postmaster_pid

# change pg_options
cat new_pg_options > $DATA_DIR/pg_options
kill -HUP $postmaster_pid

# change pg_options only for a backend
cat new_pg_options > $DATA_DIR/pg_options
kill -HUP $backend_pid
cat old_pg_options > $DATA_DIR/pg_options
```

# Chapter 31. gcc Default Optimizations

**Note:** Contributed by Brian Gallew (mailto: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 is 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 32. Backend Interface

Backend Interface (BKI) files are scripts that are input to the Postgres 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. `initdb` uses BKI files to do just that: to create a database system. However, `initdb`'s BKI files are generated internally. It generates them using the files `global.bki.source` and `local.template1.bki.source`, which it finds in the Postgres "library" directory. They get installed there as part of installing Postgres. These `.source` files get build as part of the Postgres build process, by a build program called `genbki`. `genbki` takes as input Postgres source files that double as `genbki` input that builds tables and C header files that describe those tables.

Related information may be found in documentation for `initdb`, `createdb`, and the SQL command **CREATE DATABASE**.

### BKI File Format

The Postgres backend interprets BKI files as described below. This description will be easier to understand if the `global.bki.source` file is at hand as an example. (As explained above, this `.source` file isn't quite a BKI file, but you'll be able to guess what the resulting BKI file would be anyway).

Commands are composed of a command name followed by space separated arguments. Arguments to a command which begin with a `$` are treated specially. If `$$` are the first two characters, then the first `$` is ignored and the argument is then processed normally. If the `$` is followed by space, then it is treated as a NULL value. Otherwise, the characters following the `$` are interpreted as the name of a macro causing the argument to be replaced with the macro's value. It is an error for this macro to be undefined.

Macros are defined using

```
define macro macro_name = macro_value
```

and are undefined using

```
undefine macro macro_name
```

and redefined using the same syntax as `define`.

Lists of general commands and macro commands follow.

### General Commands

`OPEN classname`

Open the class called *classname* for further manipulation.

`CLOSE [classname]`

Close the open class called *classname*. It is an error if *classname* is not already opened. If no *classname* is given, then the currently open class is closed.

**PRINT**

Print the currently open class.

**INSERT** [OID=*oid\_value*] (*value1 value2 ...*)

Insert a new instance to the open class using *value1*, *value2*, etc., for its attribute values and *oid\_value* for its OID. If *oid\_value* is not 0, then this value will be used as the instance's object identifier. Otherwise, it is an error.

**INSERT** (*value1 value2 ...*)

As above, but the system generates a unique object identifier.

**CREATE** *classname* (*name1 = type1* [,*name2 = type2*[,...]])

Create a class named *classname* with the attributes given in parentheses.

**OPEN** (*name1 = type1* [,*name2 = type2*[,...]]) **AS** *classname*

Open a class named *classname* for writing but do not record its existence in the system catalogs. (This is primarily to aid in bootstrapping.)

**DESTROY** *classname*

Destroy the class named *classname*.

**DEFINE INDEX** *indexname* **ON** *class\_name* **USING** *aname* (*opclass attr* | *function(attr)*)

Create an index named *indexname* on the class named *classname* using the *aname* access method. The fields to index are called *name1*, *name2* etc., and the operator collections to use are *collection\_1*, *collection\_2* etc., respectively.

**Note:** This last sentence doesn't reference anything in the example. Should be changed to make sense. - Thomas 1998-08-04

## Macro Commands

**DEFINE FUNCTION** *macro\_name* **AS** *rettype function\_name(args)*

Define a function prototype for a function named *macro\_name* which has its value of type *rettype* computed from the execution *function\_name* with the arguments *args* declared in a C-like manner.

**DEFINE MACRO** *macro\_name* **FROM FILE** *filename*

Define a macro named *macro\_name* which has its value read from the file called *filename*.

## Debugging Commands

**Note:** This section on debugging commands was commented-out in the original documentation. Thomas 1998-08-05

**r**

Randomly print the open class.

m -1

Toggle display of time information.

m 0

Set retrievals to now.

m 1 Jan 1 01:00:00 1988

Set retrievals to snapshots of the specified time.

m 2 Jan 1 01:00:00 1988, Feb 1 01:00:00 1988

Set retrievals to ranges of the specified times. Either time may be replaced with space if an unbounded time range is desired.

*&A classname natts name1 type1 name2 type2 . . .*

Add *natts* attributes named *name1*, *name2*, etc. of types *type1*, *type2*, etc. to the class *classname*.

*&RR oldclassname newclassname*

Rename the *oldclassname* class to *newclassname*.

*&RA classname oldattname newattname classname oldattname newattname*

Rename the *oldattname* attribute in the class named *classname* to *newattname*.

## Example

The following set of commands will create the `pg_opclass` class containing the `int_ops` collection as an object with an OID of `421`, print out the class, and then close it.

```
create pg_opclass (opcname=name)
open pg_opclass
insert oid=421 (int_ops)
print
close pg_opclass
```

# Chapter 33. Page Files

A description of the database file default page format.

This section provides an overview of the page format used by Postgres classes. 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 which is stored in Postgres classes.

## Page Structure

The following table shows how pages in both normal Postgres classes and Postgres index classes (e.g., a B-tree index) are structured.

**Table 33-1. 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 which is allocated at page initialization time and which 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 class. 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 Postgres 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.

## Files

`data/`

Location of shared (global) database files.

`data/base/`

Location of local database files.

## Bugs

The page format may change in the future to provide more efficient access to large objects.

This section contains insufficient detail to be of any assistance in writing a new access method.

# Appendix DG1. The CVS Repository

The Postgres 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 Postgres server to your local machine.

## CVS Tree Organization

**Author:** Written by Marc G. Fournier (mailto: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 1.0 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 than a `-b` option added to the command, it's the same thing.

So, to create the v6.4 release I did the following:

```
$ cd postgres
$ cvs tag -b REL6_4
```

which will create the tag and the branch for the RELEASE tree.

Now, for those with CVS access, it's too simple. 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 postgresql
cd ../CURRENT
cvs checkout -P postgresql
```

which results in two directory trees, RELEASE/postgresql and CURRENT/postgresql. 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.

## 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@postgresql.org:/usr/local/cvsroot
login
```

You will be prompted for a password; enter 'postgresql'. You should only need to do this once, since the password will be saved in .cvspass in your home directory.

3. Fetch the Postgres sources:

```
cvs -z3 -d :pserver:anoncvs@postgresql.org:/usr/local/cvsroot co
-P postgresql
```

which installs the Postgres 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 Postgres 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/>.

## Getting The Source Via CVSup

An alternative to using anonymous CVS for retrieving the Postgres source tree is CVSup. CVSup was developed by John Polstra (<mailto:jdp@polstra.com>) to distribute CVS repositories and other file trees for the FreeBSD project (<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 Postgres server due to an efficient streaming transfer protocol which only sends the changes since the last update.

## Preparing A CVSup Client System

Two directory areas are required for CVSup to do its 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 Postgres 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/
```

## 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@alumni.caltech.edu 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=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 points to 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 points to 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 Postgres ftp site (<ftp://ftp.postgresql.org/pub/CVSup/README.cvsup>) 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=postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
*default tag=.
```

```

# base directory points to where CVSup will store its 'bookmarks'
file(s)
*default base=/usr/local/pgsql

# prefix directory points to where CVSup will store the actual
distribution(s)
*default prefix=/usr/local/pgsql

# complete distribution, including all below
pgsql

# individual distributions vs 'the whole thing'
# pgsql-doc
# pgsql-perl5
# pgsql-src

```

## 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 Postgres ftp site (<ftp://postgresql.org/pub>), 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/HPUX-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. If you are running FreeBSD, install the CVSup port.
  - b. If you have another platform, check for and download the appropriate binary from the Postgres ftp site (<ftp://postgresql.org/pub>).
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
$ which cvsup
$ set path=(path to cvsup $path)
$ which cvsup
/usr/local/bin/cvsup
```

## 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 (<http://m3.polymtl.ca/m3>), who are actively maintaining the code base originally developed by the DEC Systems Research Center (<http://www.research.digital.com/SRC/modula-3/html/home.html>). 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 which may not have Modula-3 installed, try:

```
# make M3FLAGS="-DNOGUI -DSTATIC"
```

4. Install the built binary:

```
# make M3FLAGS="-DNOGUI -DSTATIC" install
```

## Appendix DG2. Documentation

The purpose of documentation is to make Postgres easier to learn, use, and extend.. The documentation set should describe the Postgres system, language, and interfaces. It should be able to answer common questions and to allow a user to find those answers on his own without resorting to mailing list support.

### Documentation Roadmap

Postgres has four primary documentation formats:

Plain text for pre-installation information.

HTML, for on-line browsing and reference.

Hardcopy (Postscript or PDF), for in-depth reading and reference.

man pages, for quick reference.

**Table DG2-1. Postgres Documentation Products**

File	Description
./COPYRIGHT	Copyright notice
./INSTALL	Installation instructions (text from sgml->rtf->text)
./README	Introductory info
./register.txt	Registration message during make
./doc/bug.template	Bug report template
./doc/postgres.tar.gz	Integrated docs (HTML)
./doc/programmer.ps.gz	Programmer's Guide (Postscript)
./doc/programmer.tar.gz	Programmer's Guide (HTML)
./doc/reference.ps.gz	Reference Manual (Postscript)
./doc/reference.tar.gz	Reference Manual (HTML)
./doc/tutorial.ps.gz	Introduction (Postscript)
./doc/tutorial.tar.gz	Introduction (HTML)
./doc/user.ps.gz	User's Guide (Postscript)
./doc/user.tar.gz	User's Guide (HTML)

There are man pages available, as well as a large number of plain-text README-type files throughout the Postgres source tree.

## The Documentation Project

Packaged documentation is available in both HTML and *Postscript* formats. These are available as part of the standard Postgres installation. We discuss here working with the documentation sources and generating documentation packages.

The documentation sources are written using SGML markup of plain text files. The purpose of DocBook SGML is to allow an author to specify the structure and content of a technical document (using the DocBook DTD), and to have a document style define how that content is rendered into a final form (e.g. using Norm Walsh's Modular Style Sheets).

See Introduction to DocBook (<http://nis-www.lanl.gov/~rosalia/mydocs/docbook-intro.html>) for a nice "quickstart" summary of DocBook features. DocBook Elements (<http://www.ora.com/homepages/dtdparse/docbook/3.0/>) provides a powerful cross-reference for features of DocBook.

This documentation set is constructed using several tools, including James Clark's jade (<http://www.jclark.com/jade/>) and Norm Walsh's Modular DocBook Stylesheets (<http://www.nwalsh.com/docbook/dsssl/>).

Currently, hardcopy is produced by importing *Rich Text Format* (RTF) output from jade into ApplixWare for minor formatting fixups, then exporting as a Postscript file.

TeX (<http://sunsite.unc.edu/pub/packages/TeX/systems/unix/>) is a supported format for jade output, but is not used at this time for several reasons, including the inability to make minor format fixes before committing to hardcopy and generally inadequate table support in the TeX stylesheets.

## Documentation Sources

Documentation sources include plain text files, man pages, and html. However, most new Postgres documentation will be written using the *Standard Generalized Markup Language* (SGML) DocBook (<http://www.ora.com/davenport/>) *Document Type Definition* (DTD). Much of the existing documentation has been or will be converted to SGML.

The purpose of SGML is to allow an author to specify the structure and content of a document (e.g. using the DocBook DTD), and to have the document style define how that content is rendered into a final form (e.g. using Norm Walsh's stylesheets).

Documentation has accumulated from several sources. As we integrate and assimilate existing documentation into a coherent documentation set, the older versions will become obsolete and will be removed from the distribution. However, this will not happen immediately, and will not happen to all documents at the same time. To ease the transition, and to help guide developers and writers, we have defined a transition roadmap.

## Document Structure

There are currently five separate documents written in DocBook. Each document has a container source document which defines the DocBook environment and other document source files. These primary source files are located in `doc/src/sgml/`, along with many of the other source files used for the documentation. The primary source files are:

postgres.sgml

This is the integrated document, including all other documents as parts. Output is generated in HTML since the browser interface makes it easy to move around all of the documentation by just clicking. The other documents are available in both HTML and hardcopy.

tutorial.sgml

The introductory tutorial, with examples. Does not include programming topics, and is intended to help a reader unfamiliar with SQL. This is the "getting started" document.

user.sgml

The User's Guide. Includes information on data types and user-level interfaces. This is the place to put information on "why".

reference.sgml

The Reference Manual. Includes Postgres SQL syntax. This is the place to put information on "how".

programming.sgml

The Programmer's Guide. Includes information on Postgres extensibility and on the programming interfaces.

admin.sgml

The Administrator's Guide. Include installation and release notes.

## Styles and Conventions

DocBook has a rich set of tags and constructs, and a suprisingly large percentage are directly and obviously useful for well-formed documentation. The Postgres documentation set has only recently been adapted to SGML, and in the near future several sections of the set will be selected and maintained as prototypical examples of DocBook usage. Also, a short summary of DocBook tags will be included below.

## SGML Authoring Tools

The current Postgres documentation set was written using a plain text editor (or emacs/psgml; see below) with the content marked up using SGML DocBook tags.

SGML and DocBook do not suffer from an oversupply of open-source authoring tools. The most common toolset is the emacs/xemacs editing package with the psgml feature extension. On some systems (e.g. RedHat Linux) these tools are provided in a typical full installation.

### emacs/psgml

emacs (and xemacs) have an SGML *major mode*. When properly configured, this will allow you to use emacs to insert tags and check markup consistency.

Put the following in your `~/ .emacs` environment file (adjusting the path names to be appropriate for your system):

```
; ***** for SGML mode (psgml)

(setq sgml-catalog-files "/usr/lib/sgml/CATALOG")
(setq sgml-local-catalogs "/usr/lib/sgml/CATALOG")

(autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t )
```

and add an entry in the same file for SGML into the (existing) definition for `auto-mode-alist`:

```
(setq
  auto-mode-alist
  '(("\\.sgml$" . sgml-mode)
    ))
```

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:
--
```

The Postgres distribution includes a parsed DTD definitions file `reference.ced`. You may find that

When using `emacs/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's 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 "-//Davenport//DTD DocBook V3.0//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"`.

## Building Documentation

GNU make is used to build documentation from the DocBook sources. There are a few environment definitions which may need to be set or modified for your installation. The Makefile looks for `doc/./src/Makefile` and (implicitly) for `doc/./src/Makefile.custom` to obtain environment information. On my system, the `src/Makefile.custom` looks like

```
# Makefile.custom
# Thomas Lockhart 1998-03-01

POSTGRES DIR= /opt/postgres/current
CFLAGS+= -m486
YFLAGS+= -v

# documentation

HSTYLE= /home/lockhart/SGML/db143.d/docbook/html
PSTYLE= /home/lockhart/SGML/db143.d/docbook/print
```

where `HSTYLE` and `PSTYLE` determine the path to `docbook.dsl` for HTML and hardcopy (print) stylesheets, respectively. These stylesheet file names are for Norm Walsh's Modular Style Sheets; if other stylesheets are used then one can define `HDSL` and `PDSL` as the full path and file name for the stylesheet, as is done above for `HSTYLE` and `PSTYLE`. On many systems, these stylesheets will be found in packages installed in `/usr/lib/sgml/`, `/usr/share/lib/sgml/`, or `/usr/local/lib/sgml/`.

HTML documentation packages can be generated from the SGML source by typing

```
% cd doc/src
% make tutorial.tar.gz
% make user.tar.gz
% make admin.tar.gz
% make programmer.tar.gz
% make postgres.tar.gz
% make install
```

These packages can be installed from the main documentation directory by typing

```
% cd doc
% make install
```

## Manpages

We use the `docbook2man` utility to convert DocBook REFENTRY pages to `*roff` output suitable for man pages. At the time of writing, the utility required patching to successfully

run on the Postgres markup, and we added a small amount of new functionality to allow setting the man page section in the output file name.

`docbook2man` is written in perl, and requires the CPAN package `SGMLSpm` to run. Also, it requires `nsgmls` to be available, which is included in the jade distribution. After installing these packages, then simply run

```
$ cd doc/src
$ make man
```

which will result in a tar file being generated in the `doc/src` directory.

### **docbook2man Installation Procedure**

1. Install the `docbook2man` package, available at <http://shell.ipoline.com/~elmert/comp/docbook2X/>
2. Install the `SGMLSpm` perl module, available from CPAN mirrors.
3. Install `nsgmls` if not already available from your jade installation.

## **Hardcopy Generation for v7.0**

The hardcopy Postscript documentation is generated by converting the SGML source code to RTF, then importing into ApplixWare-4.4.1. After a little cleanup (see the following section) the output is "printed" to a postscript file.

### **Text Hardcopy**

`INSTALL` and `HISTORY` are updated for each release. For historical reasons, these files are in plain text, but are derived from the newer SGML sources.

#### **Plain Text Generation**

Both `INSTALL` and `HISTORY` are generated from existing SGML sources. They are extracted from the same intermediate RTF file.

1. Generate RTF by typing:
 

```
% cd doc/src/sgml
% make installation.rtf
```
2. Import `installation.rtf` into Applix Words.
3. Set the page width and margins.
  - a. Adjust the page width in `File.PageSetup` to 10 inches.
  - b. Select all text. Adjust the right margin using the ruler to 9.5 inches. This will give a maximum column width of 79 characters, within the 80 columns upper limit goal.

4. Lop off the parts of the document which are not needed.  
For `INSTALL`, remove all release notes from the end of the text, except for those from the current release. For `HISTORY`, remove all text up to the release notes, preserving and modifying the title and ToC.
5. Export the result as ASCII Layout .
6. Using emacs or vi, clean up the tabular information in `INSTALL`. Remove the `mailto` URLs for the porting contributors to shrink the column heights.

## Postscript Hardcopy

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. The field can be added using vi or the following small sed procedure:

```
#!/bin/sh
# fixrtf.sh
for i in $* ; do
  mv $i $i.orig
  cat $i.orig | sed 's#\stylesheetNormal;#\stylesheet{\s0
Normal;}#' > $i
done
exit
```

where the script is adding `{\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.

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 DG2-2. Indent Formatting for Table of Contents**

Style	First Indent (inches)	Left Indent (inches)
TOC-Heading 1	0.6	0.6
TOC-Heading 2	1.0	1.0
TOC-Heading 3	1.4	1.4

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 which 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. If a bibliography is present, remove the short form reference title from each entry. The DocBook stylesheets from Norm Walsh seem to print these out, even though this is a subset of the information immediately following.
8. Save the document as native Applix Words format to allow easier last minute editing later.
9. Print the document to a file in Postscript format.
10. Compress the Postscript file using `gzip`. Place the compressed file into the `doc` directory.

## Toolsets

We have documented experience with three installation methods for the various tools that are needed to process the documentation. One is installation from RPMs on Linux, the second is installation from FreeBSD *port*, and the last is a general installation from original distributions of the individual tools. 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.

## Linux RPM Installation

The simplest installation for a RedHat-compatible Linux system uses the RPM set developed by Mark Galassi at Cygnus. It should also be possible to install from sources, as described in a subsequent section.

### Installing RPMs

1. Install RPMs (<ftp://ftp.cygnus.com/pub/home/rosalia/>) for Jade and related packages.
2. Install Norm Walsh's latest style sheets. Depending on the age of the RPMs, the latest style sheets may be substantially improved from those contained in the RPMs.
3. Update your `src/Makefile.custom` to include `HSTYLE` and `PSTYLE` definitions pointing to the style sheets.

## FreeBSD Installation

There is a full set of *ports* of the documentation tools available on FreeBSD. In fact, [postgresql.org](http://postgresql.org), on which documentation is automatically updated every evening, is a FreeBSD machine.

### Installing FreeBSD Ports

1. To build the documentation on FreeBSD a number of ports will need to be installed.

```
% cd /usr/ports/devel/gmake && make install
% cd /usr/ports/textproc/docproj && make install
% cd /usr/ports/textproc/docbook && make install
% cd /usr/ports/textproc/dsssl-docbook-modular && make install
```

2. Set environment variables to access the jade toolset.

**Note:** This was not required for the FreeBSD machine at [postgresql.org](http://postgresql.org), so you may not have to do this.

```
export SMGL_ROOT=/usr/local/share/sgml
SGML_CATALOG_FILES=/usr/local/share/sgml/jade/catalog
SGML_CATALOG_FILES=/usr/local/share/sgml/html/catalog:$SGML_CATALOG_FILES
SGML_CATALOG_FILES=/usr/local/share/sgml/iso8879/catalog:$SGML_CATALOG_FILES
SGML_CATALOG_FILES=/usr/local/share/sgml/transpec/catalog:$SGML_CATALOG_FILES
SGML_CATALOG_FILES=/usr/local/share/sgml/docbook/catalog:$SGML_CATALOG_FILES
export SGML_CATALOG_FILES
```

(this is sh/bash syntax; adjust accordingly for csh/tcsh).

3. Make needs some special arguments, or these need to be added to your `Makefile.custom`:

```
HSTYLE=/usr/local/share/sgml/docbook/dsssl/modular/html/
PSTYLE=/usr/local/share/sgml/docbook/dsssl/modular/print/
```

Of course you'll need to use `gmake` rather than just plain `'make'` to build.

## Debian Installation

There is a full set of packages of the documentation tools available for Debian.

### Installing Debian Packages

1. Install `jade`, `docbook`, and `unzip`:

```
apt-get install jade
apt-get install docbook
apt-get install docbook-stylesheets
```

2. Install the latest style sheets.

- a. Verify that `unzip` is installed, or install the package:

```
apt-get install unzip
```

- b. Grab the latest stylesheet zipballs from <http://www.nwalsh.com/docbook/dsssl> and unzip it somewhere (possibly `/usr/share`).

3. Edit `src/Makefile.custom` to add appropriate `HSTYLE` and `PSTYLE` definitions:

```
HSTYLE= /usr/share/docbook/html
PSTYLE= /usr/share/docbook/print
```

## Manual Installation of Tools

This is a brief run-through of the process of obtaining and installing the software you'll need to edit DocBook source with Emacs and process it with Norman Walsh's DSSSL style sheets to create HTML and RTF.

The easiest way to obtain the SGML and DocBook tools may be to get `sgmltools` from `sgmltools` (<http://www.sgmltools.org/>). `sgmltools` requires the GNU version of `m4`. To confirm that you have the correct version of `m4` available, try

```
gnum4 --version
```

If you install GNU `m4`, install it with the name `gnum4` and `sgmltools` will find it. After the install, you will have `sgmltools`, `jade`, and Norm Walsh's DocBook style sheets. The instructions below are for installing these tools separately.

## Prerequisites

What you need:

- A working installation of GCC 2.7.2
- A working installation of Emacs 19.19 or later
- An unzip program for Unix to unpack things

What you must fetch:

- James Clark's Jade (<ftp://ftp.jclark.com/pub/jade/>) (version 1.1 in file `jade1_1.zip` was current at the time of writing)
- DocBook version 3.0 (<http://www.ora.com/davenport/docbook/current/docbk30.zip>)
- Norman Walsh's Modular Stylesheets (<http://nwalsh.com/docbook/dsssl/>) (version 1.19 was originally used to produce these documents)
- Lennart Staflin's PSGML (<ftp://ftp.lysator.liu.se/pub/sgml/>) (version 1.0.1 in `psgml-1.0.1.tar.gz` was available at the time of writing)

Important URLs:

- The Jade web page (<http://www.jclark.com/jade/>)
- The DocBook web page (<http://www.ora.com/davenport/>)
- The Modular Stylesheets web page (<http://nwalsh.com/docbook/dsssl/>)
- The PSGML web page ([http://www.lysator.liu.se/projects/about\\_psgml.html](http://www.lysator.liu.se/projects/about_psgml.html))
- Steve Pepper's Whirlwind Guide (<http://www.infotek.no/sgmltool/guide.htm>)
- Robin Cover's database of SGML software (<http://www.sil.org/sgml/publicSW.html>)

## Installing Jade

### Installing Jade

1. Read the installation instructions at the above listed URL.
2. Unzip the distribution kit in a suitable place. The command to do this will be something like
 

```
unzip -aU jade1_1.zip
```
3. Jade is not built using GNU autoconf, so you'll need to edit a `Makefile` yourself. Since James Clark has been good enough to prepare his kit for it, it is a good idea to make a build directory (named for your machine architecture, perhaps) under the main directory of the Jade distribution, copy the file `Makefile` from the main directory into it, edit it there, and then run **make** there.
 

However, the `Makefile` does need to be edited. There is a file called `Makefile.jade` in the main directory, which is intended to be used with **make -f Makefile.jade** when building Jade (as opposed to just `SP`, the SGML parser kit that

Jade is built upon). We suggest that you don't do that, though, since there is more that you need to change than what is in `Makefile.jade`, so you'd have to edit one of them anyway.

Go through the `Makefile`, reading James' instructions and editing as needed. There are various variables that need to be set. Here is a collected summary of the most important ones, with typical values:

```
prefix = /usr/local
XDEFINES =
-DSGML_CATALOG_FILES_DEFAULT="/usr/local/share/sgml/catalog"
XLIBS = -lm
RANLIB = ranlib
srcdir = ..
XLIBDIRS = grove spgrove style
XPROGDIRS = jade
```

Note the specification of where to find the default catalog of SGML support files -- you may want to change that to something more suitable for your own installation. If your system doesn't need the above settings for the math library and the **ranlib** command, leave them as they are in the `Makefile`.

4. Type **make** to build Jade and the various SP tools.
5. Once the software is built, **make install** will do the obvious.

## Installing the DocBook DTD Kit

### Installing the DocBook DTD Kit

1. You'll want to place the files that make up the DocBook DTD kit in the directory you built Jade to expect them in, which, if you followed our suggestion above, is `/usr/local/share/sgml/`. In addition to the actual DocBook files, you'll need to have a `catalog` file in place, for the mapping of document type specifications and external entity references to actual files in that directory. You'll also want the ISO character set mappings, and probably one or more versions of HTML.

One way to install the various DTD and support files and set up the `catalog` file, is to collect them all into the above mentioned directory, use a single file named `CATALOG` to describe them all, and then create the file `catalog` as a catalog pointer to the former, by giving it the single line of content:

```
CATALOG /usr/local/share/sgml/CATALOG
```

2. The `CATALOG` file should then contain three types of lines. The first is the (optional) SGML declaration, thus:

```
SGMLDECL docbook.dcl
```

Next, the various references to DTD and entity files must be resolved. For the DocBook files, these lines look like this:

```
PUBLIC "-//Davenport//DTD DocBook V3.0//EN" docbook.dtd
PUBLIC "-//USA-DOD//DTD Table Model 951010//EN" calstbl.dtd
PUBLIC "-//Davenport//ELEMENTS DocBook Information Pool
V3.0//EN" dbpool.mod
```

```
PUBLIC "-//Davenport//ELEMENTS DocBook Document Hierarchy
V3.0//EN" dbhier.mod
PUBLIC "-//Davenport//ENTITIES DocBook Additional General
Entities V3.0//EN" dbgenent.mod
```

3. Of course, a file containing these comes with the DocBook kit. Note that the last item on each of these lines is a file name, given here without a path. You can put the files in subdirectories of your main SGML directory if you like, of course, and modify the reference in the CATALOG file. DocBook also references the ISO character set entities, so you need to fetch and install these (they are available from several sources, and are easily found by way of the URLs listed above), along with catalog entries for all of them, such as:

```
PUBLIC "ISO 8879-1986//ENTITIES Added Latin 1//EN" ISO/ISOlat1
```

Note how the file name here contains a directory name, showing that we've placed the ISO entity files in a subdirectory named `ISO`. Again, proper catalog entries should accompany the entity kit you fetch.

## Installing Norman Walsh's DSSSL Style Sheets

### Installing Norman Walsh's DSSSL Style Sheets

1. Read the installation instructions at the above listed URL.
2. To install Norman's style sheets, simply unzip the distribution kit in a suitable place. A good place to do this would be `/usr/local/share`, which places the kit in a directory tree under `/usr/local/share/docbook`. The command will be something like

```
unzip -aU db119.zip
```

3. One way to test the installation is to build the HTML and RTF forms of the *PostgreSQL User's Guide*.
  - a. To build the HTML files, go to the SGML source directory, `doc/src/sgml`, and say

```
jade -t sgml -d /usr/local/share/docbook/html/docbook.dsl
-D ../graphics postgres.sgml
```

`book1.htm` is the top level node of the output..

- b. To generate the RTF output, ready for importing into your favorite word processing system and printing, type:

```
jade -t rtf -d /usr/local/share/docbook/print/docbook.dsl
-D ../graphics postgres.sgml
```

## Installing PSGML

### Installing PSGML

1. Read the installation instructions at the above listed URL.
2. Unpack the distribution file, run configure, make and make install to put the byte-compiled files and info library in place.
3. Then add the following lines to your `/usr/local/share/emacs/site-lisp/site-start.el` file to make Emacs properly load PSGML when needed:
 

```
(setq load-path
      (cons "/usr/local/share/emacs/site-lisp/psgml" load-path))
      (autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t)
```
4. If you want to use PSGML when editing HTML too, also add this:
 

```
(setq auto-mode-alist
      (cons '("\\.s?html?\\'" . sgml-mode) auto-mode-alist))
```
5. 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.
  - a. You can set the `SGML_CATALOG_FILES` environment variable.
  - b. You can customize your PSGML installation (its manual tells you how).
  - c. You can even edit the source file `psgml.el` before compiling and installing PSGML, changing the hard-coded paths to match your own default.

## Installing JadeTeX

If you want to, you can also install JadeTeX to use TeX as a formatting backend for Jade. Note that this is still quite unpolished software, and will generate printed output that is inferior to what you get from the RTF backend. Still, it works all right, especially for simpler documents that don't use tables, and as both JadeTeX and the style sheets are under continuous improvement, it will certainly get better over time.

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, and of course JadeTeX itself. All of these can be found on your friendly neighborhood CTAN site.

JadeTeX does not at the time of writing come with much of an installation guide, but there is a `makefile` which shows what is needed. It also includes a directory `cooked`, wherein you'll find some of the macro packages it needs, but not all, and not complete -- at least last we looked.

Before building the `jadetex.fmt` format file, you'll probably want to edit the `jadetex.ltx` file, to change the configuration of Babel to suit your locality. The line to change looks something like

```
\RequirePackage[german,french,english]{babel}[1997/01/23]
```

and you should obviously list only the languages you actually need, and have configured Babel for.

With JadeTeX working, you should be able to generate and format TeX output for the PostgreSQL manuals by giving the commands (as above, in the `doc/src/sgml` directory)

```
jade -t tex -d /usr/local/share/docbook/print/docbook.dsl -D
../graphics postgres.sgml
jadetex postgres.tex
jadetex postgres.tex
dvips postgres.dvi
```

Of course, when you do this, TeX will stop during the second run, and tell you that its capacity has been exceeded. This is, as far as we can tell, because of the way JadeTeX generates cross referencing information. TeX can, of course, be compiled with larger data structure sizes. The details of this will vary according to your installation.

## Alternate Toolsets

`sgml-tools v2.x` supports `jade` and `DocBook`.

# Bibliography

Selected references and readings for SQL and Postgres.

Some white papers and technical reports from the original Postgres development team are available at the University of California Computer Science Department web site (<http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/>).

## SQL Reference Books

*The Practical SQL Handbook , Using Structured Query Language* , 3, Judith Bowman, Sandra Emerson, and Marcy Darnovsky, 0-201-44787-8, 1996, Addison-Wesley, 1996.

*A Guide to the SQL Standard , A user's guide to the standard database language SQL* , C. J. Date and Hugh Darwen, 0-201-96426-0, 1997, Addison-Wesley, 1997.

*An Introduction to Database Systems* , 6, C. J. Date, 1994, Addison-Wesley, 1994.

*Understanding the New SQL , A complete guide*, Jim Melton and Alan R. Simon, 1-55860-245-3, 1993, Morgan Kaufmann, 1993.

### Abstract

Accessible reference for SQL features.

*Principles of Database and Knowledge : Base Systems* , Jeffrey D. Ullman, Computer Science Press , 1988 .

## PostgreSQL-Specific Documentation

*The PostgreSQL Administrator's Guide* , Edited by Thomas Lockhart, 2000-05-01, The PostgreSQL Global Development Group.

*The PostgreSQL Developer's Guide* , Edited by Thomas Lockhart, 2000-05-01, The PostgreSQL Global Development Group.

*The PostgreSQL Programmer's Guide* , Edited by Thomas Lockhart, 2000-05-01, The PostgreSQL Global Development Group.

*The PostgreSQL Tutorial Introduction* , Edited by Thomas Lockhart, 2000-05-01, The PostgreSQL Global Development Group.

*The PostgreSQL User's Guide* , Edited by Thomas Lockhart, 2000-05-01, The PostgreSQL Global Development Group.

*Enhancement of the ANSI SQL Implementation of PostgreSQL* , Stefan Simkovic, O.Univ.Prof.Dr.. Georg Gottlob, November 29, 1998, Department of Information Systems, Vienna University of Technology .

Discusses SQL history and syntax, and describes the addition of INTERSECT and EXCEPT constructs into Postgres. Prepared as a Master's Thesis with the support of O.Univ.Prof.Dr. Georg Gottlob and Univ.Ass. Mag. Katrin Seyr at Vienna University of Technology.

*The Postgres95 User Manual* , A. Yu and J. Chen, The POSTGRES Group , Sept. 5, 1995, University of California, Berkeley CA.

## Proceedings and Articles

- Partial indexing in POSTGRES: research project* , Nels Olson, 1993, UCB Engin T7.49.1993 O676, University of California, Berkeley CA.
- A Unified Framework for Version Modeling Using Production Rules in a Database System* , L. Ong and J. Goh, April, 1990, ERL Technical Memorandum M90/33, University of California, Berkeley CA.
- The Postgres Data Model* , Rowe and Stonebraker, 1987 , Sept. 1987, VLDB Conference, Brighton, England, 1987.
- Generalized partial indexes*  
(<http://simon.cs.cornell.edu/home/praveen/papers/partindex.de95.ps.Z>) , P. Seshadri and A. Swami, March 1995, Eleventh International Conference on Data Engineering, 1995, Cat. No.95CH35724, IEEE Computer Society Press.
- The Design of Postgres* , M. Stonebraker and L. Rowe, May 1986, Conference on Management of Data, Washington DC, ACM-SIGMOD, 1986.
- The Design of the Postgres Rules System*, M. Stonebraker, E. Hanson, and C. H. Hong, Feb. 1987, Conference on Data Engineering, Los Angeles, CA, IEEE, 1987.
- The Postgres Storage System* , M. Stonebraker, Sept. 1987, VLDB Conference, Brighton, England, 1987.
- A Commentary on the Postgres Rules System* , M. Stonebraker, M. Hearst, and S. Potamianos, Sept. 1989, Record 18(3), SIGMOD, 1989.
- The case for partial indexes (DBMS)*  
(<http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-17.pdf>) , M. Stonebraker, Dec. 1989, Record 18(no.4):4-11, SIGMOD, 1989.
- The Implementation of Postgres* , M. Stonebraker, L. A. Rowe, and M. Hirohama, March 1990, Transactions on Knowledge and Data Engineering 2(1), IEEE.
- On Rules, Procedures, Caching and Views in Database Systems* , M. Stonebraker and et al, June 1990, Conference on Management of Data, ACM-SIGMOD.