PostgreSQL 7.1 Programmer’s Guide

The PostgreSQL Global Development Group

PostgreSQL 7.1 Programmer’s Guide

by The PostgreSQL Global Development Group

Legal Notice

Table of Contents

List of Tables

List of Figures

List of Examples

Preface

1. What is PostgreSQL?

PostgreSQL is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2 (http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html), developed at the University of California at Berkeley Computer Science Department. The POSTGRES project, led by Professor Michael Stonebraker, was sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc.

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

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

inheritance
data types
functions

Other features provide additional power and flexibility:

constraints
triggers
rules
transaction integrity

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

2. A Short History of Postgres

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

2.1. The Berkeley Postgres Project

Implementation of the Postgres DBMS began in 1986. The initial concepts for the system were presented in *The Design of Postgres* and the definition of the initial data model appeared in *The Postgres Data Model*. The design of the rule system at that time was described in *The Design of the Postgres Rules System*. The rationale and architecture of the storage manager were detailed in *The Postgres Storage System*.

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

Postgres has been used to implement many different research and production applications. These include: a financial data analysis system, a jet engine performance monitoring package, an asteroid tracking database, a medical information database, and several geographic information systems. Postgres has also been used as an educational tool at several universities. Finally, Illustra Information Technologies (http://www.illustra.com/) (since merged into Informix (http://www.informix.com/)) picked up the code and commercialized it. Postgres became the primary data manager for the Sequoia 2000 (http://www.sdsc.edu/0/Parts_Collabs/S2K/s2k_home.html) scientific computing project in late 1992.

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

2.2. Postgres95

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

Postgres95 code was completely ANSI C and trimmed in size by 25%. Many internal changes improved performance and maintainability. Postgres95 v1.0.x ran about 30-50% faster on the Wisconsin Benchmark compared to Postgres v4.2. Apart from bug fixes, these were the major enhancements:

The query language Postquel was replaced with SQL (implemented in the server). Subqueries were not supported until PostgreSQL (see below), but they could be imitated in Postgres95 with user-defined SQL functions. Aggregates were re-implemented. Support for the GROUP BY query clause was also added. The `libpq` interface remained available for C programs.

In addition to the monitor program, a new program (psql) was provided for interactive SQL queries using GNU `readline`.

A new front-end library, `libpgtcl`, supported Tcl-based clients. A sample shell, pgtclsh, provided new Tcl commands to interface tcl programs with the Postgres95 backend.

The large object interface was overhauled. The Inversion large objects were the only mechanism for storing large objects. (The Inversion file system was removed.)

The instance-level rule system was removed. Rules were still available as rewrite rules.

A short tutorial introducing regular SQL features as well as those of Postgres95 was distributed with the source code.

GNU make (instead of BSD make) was used for the build. Also, Postgres95 could be compiled with an unpatched gcc (data alignment of doubles was fixed).

2.3. PostgreSQL

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

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

Major enhancements in PostgreSQL include:

Table-level locking has been replaced with multi-version concurrency control, which allows readers to continue reading consistent data during writer activity and enables hot backups from pg_dump while the database stays available for queries.

Important backend features, including subselects, defaults, constraints, and triggers, have been implemented.

Additional SQL92-compliant language features have been added, including primary keys, quoted identifiers, literal string type coercion, type casting, and binary and hexadecimal integer input.

Built-in types have been improved, including new wide-range date/time types and additional geometric type support.

Overall backend code speed has been increased by approximately 20-40%, and backend start-up time has decreased 80% since version 6.0 was released.

3. Documentation Resources

This manual set is organized into several parts:

Tutorial

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

User’s Guide

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

Programmer’s Guide

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

Administrator’s Guide

Installation and server management information

Reference Manual

Reference pages for SQL command syntax and client and server programs

Developer’s Guide

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

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

man pages

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

FAQs

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

READMEs

README files are available for some contributed packages.

Web Site

The PostgreSQL web site (http://www.postgresql.org) carries details on the latest release, upcoming features, and other information to make your work or play with PostgreSQL more productive.

Mailing Lists

The <`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 User’s Lounge (http://www.postgresql.org/users-lounge/) section of the PostgreSQL web site for details.

Yourself!

PostgreSQL is an open source effort. As such, it depends on the user community for ongoing support. As you begin to use PostgreSQL, you will rely on others for help, either through the documentation or through the mailing lists. Consider contributing your knowledge back. If you learn something which is not in the documentation, write it up and contribute it. If you add features to the code, contribute 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@postgresql.org`> (archive (http://www.postgresql.org/mhonarc/pgsql-docs/)) mailing list is the place to get going.

4. Terminology and Notation

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

An *administrator* is generally a person who is in charge of installing and running the server. A *user* could be anyone who is using, or wants to use, any part of the PostgreSQL system. These terms should not be interpreted too narrowly; this documentation set does not have fixed presumptions about system administration procedures.

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

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

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

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

5. Bug Reporting Guidelines

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

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

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

5.1. Identifying Bugs

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

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

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

Being slow or resource-hogging is not necessarily a bug. Read the documentation or ask on one of the mailing lists for help in tuning your applications. Failing to comply to 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 cannot decode the information on the TODO list, report your problem. The least we can do is make the TODO list clearer.

5.2. What to report

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

The following items should be contained in every bug report:

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

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

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

> **Note:** In case of fatal errors, the error message provided by the client might not contain all the information available. In that case, also look at the log 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 should not have to spend the time to decode the exact semantics behind your commands. Especially refrain from merely saying that "This is not what SQL says/Oracle does." Digging out the correct behavior from SQL is not a fun undertaking, nor do we all know how all the other relational databases out there behave. (If your problem is a program crash you can obviously omit this item.)

Any command line options and other start-up options, including concerned environment variables or configuration files that you changed from the default. Again, be exact. If you are using a 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 the version of the server you are connected to. Most executable programs also support a `--version` option; at least `postmaster --version` and `psql --version` should work. If the function or the options do not exist then your version is probably old enough. You can also look into the `README` file in the source directory or at the name of your distribution file or package name. If you run a 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.

If your version is older than 7.1 we will almost certainly tell you to upgrade. There are tons of bug fixes in each new release, that is why we make new releases.

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

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

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

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

5.3. Where to report bugs

In general, send bug reports to the bug report mailing list at <`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`> or <`pgsql-general@postgresql.org`>. These mailing lists are for answering user questions and their subscribers normally do not wish to receive bug reports. More importantly, they are unlikely to fix them.

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

If you have a problem with the documentation, send email to the documentation mailing list <`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`>, so we (and you) can work on porting PostgreSQL to your platform.

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

6. Y2K Statement

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

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

The author of this statement, a volunteer on the 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* 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.

Organization

The first part of this manual is the description of the client-side programming interfaces and support libraries for various languages. The second part explains the PostgreSQL approach to extensibility and describe how users can extend PostgreSQL by adding user-defined types, operators, aggregates, and both query language and programming language functions. After a discussion of the PostgreSQL rule system, we discuss the trigger and SPI interfaces. The third part documents the procedural languages available in the PostgreSQL distribution.

Proficiency with Unix and C programming is assumed.

I. Client Interfaces

Chapter 1. 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++), `libpgtcl` (Tcl), Perl, and `ecpg`. So some aspects of libpq's behavior will be important to you if you use one of those packages.

Three short programs are included at the end of this section to show how to write programs that use `libpq`. There are several complete examples of `libpq` applications in the following directories:

```
../src/test/regress
../src/test/examples
../src/bin/psql
```

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

1.1. 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 prefered 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 this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is the name of the directory in which the socket file is stored. The default is to connect to a Unix-domain socket in `/tmp`.

`hostaddr`

IP address of host to connect to. This should be in standard numbers-and-dots form, as used by the BSD functions inet_aton et al. If a 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 that is not the name of the machine at hostaddr.

Without either a host name or host address, libpq will connect using a local Unix domain socket.

`port`

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

`requiressl`

Set to ’1’ to require SSL connection to the backend. Libpq will then refuse to connect if the server does not support SSL. Set to ’0’ (default) to negotiate with server.

If any parameter is unspecified, then the corresponding environment variable (see "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 start-up.

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.

`PQconndefaults` Returns the default connection options.

```
PQconninfoOption *PQconndefaults(void)

struct PQconninfoOption
{
    char   *keyword;   /* The keyword of the option */
    char   *envvar;    /* Fallback environment variable name */
    char   *compiled;  /* Fallback compiled in default value */
    char   *val;       /* Option's current value, or NULL */
    char   *label;     /* Label for field in connect dialog */
    char   *dispchar;  /* Character to display for this field
                          in a connect dialog. Values are:
                          ""        Display entered value as is
                          "*"       Password field - hide value
                          "D"       Debug option - don't show by default */
    int     dispsize;  /* Field size in characters for dialog */
}
```

Returns a connection options array. This may be used to determine all possible PQconnectdb options and their current default values. The return value points to an array of PQconninfoOption structs, which ends with an entry having a NULL keyword pointer. Note that the default values ("val" fields) will depend on environment variables and other context. Callers must treat the connection options data as read-only.

After processing the options array, free it by passing it to PQconninfoFree(). If this is not done, a small amount of memory is leaked for each call to PQconndefaults().

In Postgres versions before 7.0, PQconndefaults() 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!

`PQgetssl` Returns the SSL structure used in the connection, or NULL if SSL is not in use.

```
SSL *PQgetssl(const PGconn *conn);
```

This structure can be used to verify encryption levels, check server certificate and more. Refer to the OpenSSL documentation for information about this structure.

You must define `USE_SSL` in order to get the prototype for this function. Doing this will also automatically include `ssl.h` from OpenSSL.

`PQgetssl` Returns the SSL structure used in the connection, or NULL if SSL is not in use.

```
SSL *PQgetssl(const PGconn *conn);
```

This structure can be used to verify encryption levels, check server certificate and more. Refer to the OpenSSL documentation for information about this structure.

You must define `USE_SSL` in order to get the prototype for this function. Doing this will also automatically include `ssl.h` from OpenSSL.

1.2. 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 (rows) 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_index);
```

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.

1.3. 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 nonblocking status of the connection.

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

Sets the state of the connection to nonblocking if arg is TRUE, blocking if arg is FALSE. Returns 0 if OK, -1 if error.

In the nonblocking state, calls to `PQputline`, `PQputnbytes`, `PQsendQuery` and `PQendcopy` will not block but instead return an error if they need to be called again.

When a database connection has been set to 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 >= 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.

1.4. 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.

1.5. 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.

1.6. 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 `PGgetline`. It must be issued or the backend may get "out of sync" with the frontend. Upon return from this function, the backend is ready to receive the next 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 double precision)");
PQexec(conn, "copy foo from stdin");
PQputline(conn, "3\thello world\t4.5\n");
PQputline(conn,"4\tgoodbye world\t7.11\n");
...
PQputline(conn,"\\.\n");
PQendcopy(conn);
```

When using `PQgetResult`, the application should respond to a `PGRES_COPY_OUT` result by executing `PQgetline` repeatedly, followed by `PQendcopy` after the terminator line is seen. It should then return to the `PQgetResult` loop until `PQgetResult` returns NULL. Similarly a `PGRES_COPY_IN` result is processed by a series of `PQputline` calls followed by `PQendcopy`, then return to the `PQgetResult` loop. This arrangement will ensure that a copy in or copy out command embedded in a series of SQL commands will be executed correctly.

Older applications are likely to submit a copy in or copy out via `PQexec` and assume that the transaction is done after `PQendcopy`. This will work correctly only if the copy in/out is the only SQL command in the query string.

1.7. libpq Tracing Functions

`PQtrace` Enable tracing of the frontend/backend communication to a debugging file stream.

```
void PQtrace(PGconn *conn
             FILE *debug_port)
```

`PQuntrace` Disable tracing started by PQtrace

```
void PQuntrace(PGconn *conn)
```

1.8. libpq Control Functions

`PQsetNoticeProcessor` Control reporting of notice and warning messages generated by libpq.

```
typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

By default, libpq prints "notice" messages from the backend on `stderr`, as well as a few error messages that it generates by itself. This behavior can be overridden by supplying a callback function that does something else with the messages. The callback function is passed the text of the error message (which includes a trailing newline), plus a void pointer that is the same one passed to `PQsetNoticeProcessor`. (This pointer can be used to access application-specific state if needed.) The default notice processor is simply

```
static void
defaultNoticeProcessor(void * arg, const char * message)
{
    fprintf(stderr, "%s", message);
}
```

To use a special notice processor, call `PQsetNoticeProcessor` just after creation of a new PGconn object.

The return value is the pointer to the previous notice processor. If you supply a callback function pointer of NULL, no action is taken, but the current pointer is returned.

Once you have set a notice processor, you should expect that that function could be called as long as either the PGconn object or PGresult objects made from it exist. At creation of a PGresult, the PGconn's current notice processor pointer is copied into the PGresult for possible use by routines like `PQgetvalue`.

1.9. Environment Variables

The following environment variables can be used to select default connection parameter values, which will be used by `PQconnectdb` or `PQsetdbLogin` if no value is directly specified by the calling code. These are useful to avoid hard-coding database names into simple application programs.

PGHOST sets the default server name. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is the name of the directory in which the socket file is stored (default "/tmp").

PGPORT sets the default TCP port number or Unix-domain socket file extension for communicating with the 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:

PGGEQO sets the default mode for the genetic optimizer.

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

1.10. 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.

1.11. Example Programs

Example 1-1. libpq Example Program 1

```
/*
 * testlibpq.c
 *
 * Test the C version of libpq, the PostgreSQL frontend
 * library.
 */
#include <stdio.h>
#include <libpq-fe.h>

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;

    /* FILE *debug; */

    PGconn     *conn;
    PGresult   *res;

    /*
     * begin, by setting the parameters for a backend connection if the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend server */
    dbName = "template1";

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database ’%s’ failed.\n", dbName);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* debug = fopen("/tmp/trace.out","w"); */
    /* PQtrace(conn, debug);  */

    /* start a transaction block */
    res = PQexec(conn, "BEGIN");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to avoid
     * memory leaks
     */
```

```
    PQclear(res);

    /*
     * fetch rows from the pg_database, the system catalog of
     * databases
     */
     res = PQexec(conn, "DECLARE mycursor CURSOR FOR select * from
pg_database");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);
    res = PQexec(conn, "FETCH ALL in mycursor");
    if (!res || PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL command didn’t return tuples properly\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /* first, print out the attribute names */
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");

    /* next, print out the rows */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }
    PQclear(res);

    /* close the cursor */
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);

    /* commit the transaction */
    res = PQexec(conn, "COMMIT");
    PQclear(res);

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

    /* fclose(debug); */
    return 0;

}
```

Example 1-2. libpq Example Program 2

```
/*
 * testlibpq2.c
 *  Test of the asynchronous notification interface
 *
 * Start this program, then from psql in another window do
 *   NOTIFY TBL2;
 *
 * Or, if you want to get fancy, try this:
 * Populate a database with the following:
 *
 *   CREATE TABLE TBL1 (i int4);
 *
 *   CREATE TABLE TBL2 (i int4);
 *
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 values (new.i); NOTIFY TBL2);
 *
 * and do
 *
 *   INSERT INTO TBL1 values (10);
 *
 */
#include <stdio.h>
#include "libpq-fe.h"

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;

    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;

    /*
     * begin, by setting the parameters for a backend connection if the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend server */
    dbName = getenv("USER");    /* change this to the name of your test
                                 * database */

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database ’%s’ failed.\n", dbName);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "LISTEN TBL2");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "LISTEN command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to avoid
     * memory leaks
     */
    PQclear(res);

    while (1)
    {

        /*
         * wait a little bit between checks; waiting with select()
         * would be more efficient.
         */
        sleep(1);
        /* collect any asynchronous backend messages */
        PQconsumeInput(conn);
        /* check for asynchronous notify messages */
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                 "ASYNC NOTIFY of ’%s’ from backend pid ’%d’ received\n",
                    notify->relname, notify->be_pid);
            free(notify);
        }
    }

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

    return 0;
}
```

Example 1-3. libpq Example Program 3

```
/*
 * testlibpq3.c Test the C version of Libpq, the Postgres frontend
 * library. tests the binary cursor interface
 *
 *
 *
 * populate a database by doing the following:
 *
 * CREATE TABLE test1 (i int4, d real, p polygon);
 *
 * INSERT INTO test1 values (1, 3.567, polygon '(3.0, 4.0, 1.0, 2.0)');
 *
 * INSERT INTO test1 values (2, 89.05, polygon '(4.0, 3.0, 2.0, 1.0)');
 *
 * the expected output is:
 *
 * tuple 0: got i = (4 bytes) 1, d = (4 bytes) 3.567000, p = (4
 * bytes) 2 points   boundbox = (hi=3.000000/4.000000, lo =
 * 1.000000,2.000000) tuple 1: got i = (4 bytes) 2, d = (4 bytes)
 * 89.050003, p = (4 bytes) 2 points   boundbox =
 * (hi=4.000000/3.000000, lo = 2.000000,1.000000)
 *
 *
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "utils/geo-decls.h"    /* for the POLYGON type */

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;
    int         i_fnum,
                d_fnum,
                p_fnum;
    PGconn     *conn;
    PGresult   *res;

    /*
     * begin, by setting the parameters for a backend connection if the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend server */

    dbName = getenv("USER");    /* change this to the name of your test
                                 * database */

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database ’%s’ failed.\n", dbName);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* start a transaction block */
    res = PQexec(conn, "BEGIN");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to avoid
     * memory leaks
     */
    PQclear(res);
```

```
    /*
     * fetch rows from the pg_database, the system catalog of
     * databases
     */
     res = PQexec(conn, "DECLARE mycursor BINARY CURSOR FOR select * from
test1");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "FETCH ALL in mycursor");
    if (!res || PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL command didn’t return tuples properly\n");
        PQclear(res);
        exit_nicely(conn);
    }

    i_fnum = PQfnumber(res, "i");
    d_fnum = PQfnumber(res, "d");
    p_fnum = PQfnumber(res, "p");

    for (i = 0; i < 3; i++)
    {
        printf("type[%d] = %d, size[%d] = %d\n",
               i, PQftype(res, i),
               i, PQfsize(res, i));
    }
    for (i = 0; i < PQntuples(res); i++)
    {
        int         *ival;
        float       *dval;
        int          plen;
        POLYGON     *pval;

        /* we hard-wire this to the 3 fields we know about */
        ival = (int *) PQgetvalue(res, i, i_fnum);
        dval = (float *) PQgetvalue(res, i, d_fnum);
        plen = PQgetlength(res, i, p_fnum);

        /*
         * plen doesn’t include the length field so need to
         * increment by VARHDSZ
         */
        pval = (POLYGON *) malloc(plen + VARHDRSZ);
        pval->size = plen;
        memmove((char *) &pval->npts, PQgetvalue(res, i, p_fnum), plen);
        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d,\n",
               PQgetlength(res, i, i_fnum), *ival);
        printf(" d = (%d bytes) %f,\n",
               PQgetlength(res, i, d_fnum), *dval);
         printf("  p  =  (%d  bytes)  %d  points  \tboundbox  =  (hi=%f/%f,  lo  =
%f,%f)\n",
               PQgetlength(res, i, d_fnum),
               pval->npts,
               pval->boundbox.xh,
               pval->boundbox.yh,
               pval->boundbox.xl,
               pval->boundbox.yl);
    }
    PQclear(res);

    /* close the cursor */
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);

    /* commit the transaction */
    res = PQexec(conn, "COMMIT");
    PQclear(res);

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

    return 0;
}
```

Chapter 2. Large Objects

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.

2.1. 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.) Since PostgreSQL 7.1 all large objects are placed in one system table called pg_largeobject.

2.2. 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.

2.3. 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 6.5, 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.

2.3.1. 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);
```

2.3.2. 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.

2.3.3. Exporting a Large Object

To export a large object into UNIX file, call

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

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

2.3.4. Opening an Existing Large Object

To open an existing large object, call

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

The lobjId argument specifies the Oid of the large object to open. The mode bits control whether the object is opened for reading INV_READ), writing or both. A large object cannot be opened before it is created. `lo_open` returns a large object descriptor for later use in `lo_read`, `lo_write`, `lo_lseek`, `lo_tell`, and `lo_close`.

2.3.5. Writing Data to a Large Object

The routine

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len)
```

writes len bytes from buf to large object fd. The fd argument must have been returned by a previous `lo_open`. The number of bytes actually written is returned. In the event of an error, the return value is negative.

2.3.6. Reading Data from a Large Object

The routine

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len)
```

reads len bytes from large object fd into byf. The fd argument must have been returned by a previous `lo_open`. The number of bytes actually read is returned. In the event of an error, the return value is negative.

2.3.7. Seeking on a Large Object

To change the current read or write location on a large object, call

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence)
```

This routine moves the current location pointer for the large object described by fd to the new location specified by offset. The valid values for whence are SEEK_SET, SEEK_CUR, and SEEK_END.

2.3.8. Closing a Large Object Descriptor

A large object may be closed by calling

```
int lo_close(PGconn *conn, int fd)
```

where fd is a large object descriptor returned by `lo_open`. On success, `lo_close` returns zero. On error, the return value is negative.

2.3.9. Removing a Large Object

To remove a large object from the database, call

```
Oid lo_unlink(PGconn *conn, Oid lobjId)
```

The lobjId argument specifies the Oid of the large object to remove.

2.4. 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';
```

2.5. 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.

2.6. 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   1995/09/01
23:55:00 jolly Exp
 *
 *-------------------------------------------------------------
 */
#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
"lob
jOid"
 *
 */
Oid
importFile(PGconn *conn, char *filename)
{
    Oid         lobjId;
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * open the file to be read in
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "can’t open unix file %s\n", filename);
    }

    /*
     * create the large object
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "can’t create large object\n");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading large object\n");
    }

    (void) close(fd);
    (void) lo_close(conn, lobj_fd);

    return lobjId;
}

void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can’t open large object %d\n",
                lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = ’ ’;
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nwritten;
    int         i;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can’t open large object %d\n",
                lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = ’X’;
    buf[i] = ’ ’;

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

/*
```

```
 * exportFile *    export large object "lobjOid" to file "out_filename"
 *
 */
void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * create an inversion "object"
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }

    /*
     * open the file to be written to
     */
    fd = open(filename, O_CREAT | O_WRONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "can't open unix file %s\n",
                filename);
    }

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing %s\n",
                    filename);
        }
    }

    (void) lo_close(conn, lobj_fd);
    (void) close(fd);

    return;
}

void
exit_nicely(PGconn *conn)
```

```
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char       *in_filename,
               *out_filename;
    char       *database;
    Oid         lobjOid;
    PGconn     *conn;
    PGresult   *res;

    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s database_name in_filename out_filename\n",
                argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * set up the connection
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* check to see that the backend connection was successfully made */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database ’%s’ failed.\n", database);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "begin");
    PQclear(res);

    printf("importing file %s\n", in_filename);
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
/*
    printf("as large object %d.\n", lobjOid);

    printf("picking out bytes 1000-2000 of the large object\n");
    pickout(conn, lobjOid, 1000, 1000);

    printf("overwriting bytes 1000-2000 of the large object with X’s\n");
    overwrite(conn, lobjOid, 1000, 1000);
*/

    printf("exporting large object to file %s\n", out_filename);
/*    exportFile(conn, lobjOid, out_filename); */
    lo_export(conn, lobjOid, out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
```

Chapter 3. libpq++ - C++ Binding Library

`libpq++` is the C++ API to Postgres. `libpq++` is a set of classes that 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.

3.1. Control and Initialization

3.1.1. Environment Variables

The following environment variables can be used to set up default values for an environment and to avoid hard-coding database names into an application program:

> **Note:** Refer to Section 1.9 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 this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is the name of the directory in which the socket file is stored (default "/tmp").

PGPORT sets the default TCP port number or Unix-domain socket file extension for communicating with the 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.

3.2. libpq++ Classes

3.2.1. Connection Class: `PgConnection`

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

3.2.2. Database Class: `PgDatabase`

The database class provides C++ objects that have a connection to a backend server. To create such an object one first needs the apropriate environment for the backend to access. The following constructors deal with making a connection to a backend server from a C++ program.

3.3. 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 succeded 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.

3.4. 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 (rows) 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` PQfnumber 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()
```

3.5. 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 asyncronous 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 eturns 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.

3.6. 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 double precision)");
data.Exec("copy foo from stdin");
data.PutLine("3\tHello World\t4.5\n");
data.PutLine("4\tGoodbye World\t7.11\n");
&...
data.PutLine("\\.\n");
data.EndCopy();
```

Chapter 4. pgtcl - TCL Binding Library

`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.

4.1. Commands

Table 4-1. `pgtcl` Commands

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

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.

4.2. 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
}
```

4.3. 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

XXX thomas 1997-12-24

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 5. libpgeasy - Simplified C Library

Author: Written by Bruce Momjian (<`pgman@candle.pha.pa.us`>) and last updated 2000-03-30

pgeasy allows you to cleanly interface to the libpq library, more like a 4GL SQL interface.

It consists of set of simplified C functions that encapsulate the functionality of libpq. The functions are:

- PGresult *doquery(char *query);
- PGconn *connectdb(char *options);
- void disconnectdb();
- int fetch(void *param,...);
- int fetchwithnulls(void *param,...);
- void reset_fetch();
- void on_error_continue();
- void on_error_stop();
- PGresult *get_result();
- void set_result(PGresult *newres);
- void unset_result(PGresult *oldres);

Many functions return a structure or value, so you can do more work with the result if required.

You basically connect to the database with `connectdb`, issue your query with `doquery`, fetch the results with `fetch`, and finish with `disconnectdb`.

For `SELECT` queries, `fetch` allows you to pass pointers as parameters, and on return the variables are filled with data from the binary cursor you opened. These binary cursors 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 6. ecpg - Embedded SQL in C

This describes an embedded SQL in C package for Postgres. It was written by Linus Tolke (<`linus@epact.se`>) and Michael Meskes (<`meskes@debian.org`>). The package is installed with the Postgres distribution.

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

6.1. 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.

6.2. 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.

6.3. How To Use ecpg

This section describes how to use the ecpg tool.

6.3.1. Preprocessor

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

6.3.2. Library

The ecpg library is called `libecpg.a` or `libecpg.so`. Additionally, the library uses the `libpq` library for communication to the 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.

6.3.3. 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 sqlext[8];
} sqlca;
```

If an error occured in the last SQL statement then *`sqlca.sqlcode`* will be non-zero. If *`sqlca.sqlcode`* is less that 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 an `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 quering cannot be found or we have gone through the cursor.

6.4. 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.

6.5. 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 (meskes@debian.org). 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.

6.6. 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.

6.6.1. ToDo List

This version the preprocessor has some flaws:

Library functions

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

Structures ans 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.

6.6.2. 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 chararcter string.

Disconnect statements

A disconnect statement looks loke:

```
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)

6.6.3. 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.)

6.6.4. 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 where 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 7. ODBC Interface

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

ODBC (Open Database Connectivity) is an abstract API that allows you to write applications that can interoperate with various RDBMS servers. ODBC provides a product-neutral interface between frontend applications and database servers, allowing a user or developer to write applications that are transportable between servers from different manufacturers..

7.1. 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 specifc 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 that have exactly the same data. Using ODBC, your Windows application would make exactly the same calls and the back end data source would look the same (to the Windows app).

7.2. Installation

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 free ODBC driver for Unix called iODBC which can be obtained via http://www.iodbc.org. Instructions for installing iODBC are contained in the iODBC distribution. Having said that, any driver manager that you can find for your platform should support the psqlODBC driver, or any other ODBC driver for that matter.

To install psqlODBC you simply need to supply the `--enable-odbc` option to the `configure` script when you are building the entire PostgreSQL distribution. The library and header files will then automatically be built and installed with the rest of the programs. If you forget that option or want to build the ODBC driver later you can change into the directory `src/interfaces/odbc` and do `make` and `make install` there.

The installation-wide configuration file `odbcinst.ini` will be installed into the directory `/usr/local/pgsql/etc/`, or equivalent, depending on what `--prefix` and/or `--sysconfdir` options you supplied to `configure`. Since this file can also be shared between different ODBC drivers you can also install it in a shared location. To do that, override the location of this file with the `--with-odbcinst` option.

Additionally, you should install the ODBC catalog extensions. That will provide a number of functions mandated by the ODBC standard that are not supplied by PostgreSQL by default. The file `/usr/local/pgsql/share/odbc.sql` (in the default installation layout) contains the appropriate definitions, which you can install as follows:

```
psql -d template1 -f LOCATION/odbc.sql
```

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

7.2.1. 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.

7.3. 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 = prefix/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
```

7.4. Windows Applications

In the real world, differences in drivers and the level of ODBC support lessens the potential of ODBC:

Access, Delphi, and Visual Basic all support ODBC directly.

Under C++, such as Visual C++, you can use the C++ ODBC API.

In Visual C++, you can use the CRecordSet class, which wraps the ODBC API set within an MFC 4.2 class. This is the easiest route if you are doing Windows C++ development under Windows NT.

7.4.1. Writing Applications

If I write an application for 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.

7.5. ApplixWare

ApplixWare has an ODBC database interface supported on at least some platforms. ApplixWare 4.4.2 has been demonstrated under Linux with Postgres 7.0 using the psqlODBC driver contained in the Postgres distribution.

7.5.1. 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.

7.5.2. Common Problems

The following messages can appear while trying to make an ODBC connection through Applix Data:

Cannot launch gateway on server

`elfodbc` can't find `libodbc.so`. Check your `axnet.cnf`.

Error from ODBC Gateway: IM003::[iODBC][Driver Manager]Specified driver could not be loaded

`libodbc.so` cannot find the driver listed in `.odbc.ini`. Verify the settings.

Server: Broken Pipe

The driver process has terminated due to some other problem. You might not have an up-to-date version of the 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 (<`scampbell@lear.com`>).

Author: Contributed by Steve Campbell (<`scampbell@lear.com`>), 1998-10-20

The axnet program's security system seems a little suspect. axnet does things on behalf of the user and on a true 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.

7.5.3. Debugging ApplixWare ODBC Connections

One good tool for debugging connection problems uses the Unix system utility strace.

Debugging with strace

1. Start applixware.
2. Start an strace on the axnet process. For example, if

```
% ps -aucx | grep ax
```

shows

```
cary   10432  0.0  2.6  1740   392  ?  S  Oct  9  0:00 axnet
cary   27883  0.9 31.0 12692  4596  ?  S   10:24  0:04 axmain
```

Then run

```
% strace -f -s 1024 -p 10432
```

3. Check the strace output.

   Note from Cary: Many of the error messages from ApplixWare go to `stderr`, but I'm not sure where `stderr` is sent, so strace is the way to find out.

For example, after getting a "`Cannot launch gateway on server`", I ran strace on axnet and got

```
[pid 27947] open("/usr/lib/libodbc.so", O_RDONLY) = -1 ENOENT
(No such file or directory)
[pid 27947] open("/lib/libodbc.so", O_RDONLY) = -1 ENOENT
(No such file or directory)
[pid 27947] write(2, "/usr2/applix/axdata/elfodbc:
can't load library 'libodbc.so'\n", 61) = -1 EIO (I/O error)
```

So what is happening is that applix elfodbc is searching for libodbc.so, but it can't find it. That is why axnet.cnf needed to be changed.

7.5.4. Running the ApplixWare Demo

In order to go through the *ApplixWare Data Tutorial*, you need to create the sample tables that the Tutorial refers to. The ELF Macro used to create the tables tries to use a NULL condition on many of the database columns, and 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.

7.5.5. Useful Macros

You can add information about your database login and password to the standard Applix start-up macro file. This is an example `~/axhome/macros/login.am` file:

```
macro login
set_set_system_var@("sql_username@","tgl")
set_system_var@("sql_passwd@","no$way")
endmacro
```

Caution

You should be careful about the file protections on any file containing username and password information.

Chapter 8. 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 system's own network protocol. Because of this, the driver is platform independent; once compiled, the driver can be used on any system.

This chapter is not intended as a complete guide to JDBC programming, but should help to get you started. For more information refer to the standard JDBC API documentation. Also, take a look at the examples included with the source. The basic example is used here.

8.1. Setting up the JDBC Driver

8.1.1. Building the Driver

Precompiled versions of the driver are regularly made available on the PostgreSQL JDBC web site (http://jdbc.postgresql.org). Here we describe how to build the driver manually.

Starting with PostgreSQL version 7.1, the JDBC driver is built using Ant, a special tool for building Java-based packages. You should download Ant from the Ant web site (http://jakarta.apache.org/ant/index.html) and install it before proceeding. Precompiled Ant distributions are typically set up to read a file `.antrc` in the current user's home directory for configuration. For example, to use a different JDK than the default, this may work:

```
JAVA_HOME=/usr/local/sun-jdk1.3
JAVACMD=$JAVA_HOME/bin/java
```

The build the driver, add the `--with-java` option to your `configure` command line, e.g.,

```
$ ./configure --prefix=xxx --with-java ...
```

This will build and install the driver along with the rest of the PostgreSQL package when you issue the `gmake` and `gmake install` commands. If you only want to build the driver and not the rest of PostgreSQL, change into the directory `src/interfaces/jdbc` and issue the respective `make` command there. Refer to the PostgreSQL installation instructions for more information about the configuration and build process.

Note: Do not try to build by calling **javac** directly, as the driver uses some dynamic loading techniques for performance reasons, and **javac** cannot cope. Do not try to run **ant** directly either, because some configuration information is communicated through the makefiles. Running **ant** directly without providing these parameters will result in a broken driver.

8.1.2. Setting up the Class Path

To use the driver, the jar archive `postgresql.jar` needs to be included in the class path, either by putting it in the CLASSPATH environment variable, or by using flags on the **java** command line. By default, the jar archive is installed in the directory `/usr/local/pgsql/share/java`. You may have it in a different directory if you used the `--prefix` option when you ran `configure`.

For instance, I have an application that uses the JDBC driver to access a large database containing astronomical objects. I have the application and the JDBC driver installed in the `/usr/local/lib` directory, and the Java JDK installed in `/usr/local/jdk1.1.6`. To run the application, I would use:

```
export
CLASSPATH=/usr/local/lib/finder.jar:/usr/local/pgsql/share/java/postgresql.j-
ar:.
java uk.org.retep.finder.Main
```

`finder.jar` contains my application.

Loading the driver from within the application is covered in Section 8.2.

8.1.3. Preparing the Database for JDBC

Because Java can only use TCP/IP connections, the Postgres server must be configured to accept TCP/IP connections, for instance by supplying the `-i` option flag when starting the **postmaster**.

Also, the client authentication setup in the `pg_hba.conf` file may need to be configured. Refer to the *Administrator's Guide* for details. The JDBC Driver supports trust, ident, password, and crypt authentication methods.

8.2. Using the Driver

8.2.1. Importing JDBC

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

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

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

8.2.2. Loading the Driver

Before you can connect to a database, you need to load the driver. There are two methods available, and it depends on your code which is the best one to use.

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

```
Class.forName("org.postgresql.Driver");
```

This will load the driver, and while loading, the driver will automatically register itself with JDBC.

> **Note:** The `forName()` method can throw a `ClassNotFoundException` if the driver is not available.

This is the most common method to use, but restricts your code to use just Postgres. If your code may access another database system in the future, and you do not use any Postgres-specific extensions, then the second method is advisable.

The second method passes the driver as a parameter to the JVM as it starts, using the `-D` argument. Example:

```
java -Djdbc.drivers=org.postgresql.Driver example.ImageViewer
```

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

Now, this method is the better one to use because it allows your code to be used with other database packages without recompiling the code. The only thing that would also change is the connection URL, which is covered next.

One last thing: When your code then tries to open a `Connection`, and you get a No driver available `SQLException` being thrown, this is probably caused by the driver not being in the class path, or the value in the parameter not being correct.

8.2.3. Connecting to the Database

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

- `jdbc:postgresql:`*`database`*
- `jdbc:postgresql:`*`//host/database`*
- `jdbc:postgresql:`*`//host`*`:`*`port/database`*

where:

`host`

The host name of the server. Defaults to `localhost`.

`port`

The port number the server is listening on. Defaults to the 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, username, password);
```

8.2.4. Closing the Connection

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

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

8.3. Issuing a Query and Processing the Result

Any time you want to issue SQL statements to the database, you require a `Statement` 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. Example 8-1 illustrates this process.

Example 8-1. Processing a Simple Query in JDCB

This example with issue a simple query and print out the first column of each row.

```
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();
```

8.3.1. Using the `Statement` Interface

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

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

If you need to perform a query while processing a `ResultSet`, you can simply create and use another `Statement`.

If you are using threads, and several are using the database, you must use a separate `Statement` for each thread. Refer to Section 8.7 if you are thinking of using threads, as it covers some important points.

8.3.2. Using the `ResultSet` Interface

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

- Before reading any values, you must call `next()`. This returns true if there is a result, but more importantly, it prepares the row for processing.
- Under the JDBC specification, you should access a field only once. It is safest to stick to this rule, although at the current time, the 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 using it.
- Once you make another query with the `Statement` used to create a `ResultSet`, the currently open `ResultSet` instance is closed automatically.

8.4. 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 int, b int)");
```

8.5. 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 in a separate table in a special format, and are referred to from your own tables by an OID value.

Important: For Postgres, you must access Large Objects within an SQL transaction. You would open a transaction by using the `setAutoCommit()` method with an input parameter of `false`:

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

There are two methods of using Large Objects. The first is the standard JDBC way, and is documented here. The other, uses PostgreSQL extensions 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 Large Objects 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. Example 8-2 illustrates the usage of this approach.

Example 8-2. Using the JDBC Large Object Interface

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

```
CREATE TABLE images (imgname text, 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();
```

The question marks must appear literally. The actual data is substituted by the next lines.

Here, `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. Notice that the creation of the Large Object itself in the database happens transparently.

Retrieving an image is even easier. (We use `PreparedStatement` here, but the `Statement` class 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();
```

Here you can see how the Large Object is retrieved as an `InputStream`. You will 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.

8.6. PostgreSQL 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. As these are facilities unique to Postgres, we support them from Java, with a set of extension API’s. Some features within the core of the standard driver actually use these extensions to implement Large Objects, etc.

8.6.1. Accessing the Extensions

To access some of the extensions, you need to use some extra methods in the `org.postgresql.Connection` class. In this case, you would need to case the return value of `Driver.getConnection()`. For example:

```
Connection db = Driver.getConnection(url, username, password);
// ...
// later on
Fastpath fp = ((org.postgresql.Connection)db).getFastpathAPI();
```

8.6.1.1. Class `org.postgresql.Connection`

```
public class Connection extends Object implements Connection

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

These are the extra methods used to gain access to PostgreSQL’s extensions. Methods defined by `java.sql.Connection` are not listed.

8.6.1.1.1. Methods

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

This returns the Fastpath API for the current connection. It is primarily used by the Large Object API.

The best way to use this is as follows:

```
import org.postgresql.fastpath.*;
...
Fastpath fp = ((org.postgresql.Connection)myconn).getFastpathAPI();
```

where myconn is an open Connection to PostgreSQL.

Returns: Fastpath object allowing access to functions on the PostgreSQL backend.

Throws: SQLException by Fastpath when initializing for first time

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

This returns the Large Object API for the current connection.

The best way to use this is as follows:

```
import org.postgresql.largeobject.*;
...
LargeObjectManager                            lo                             =
((org.postgresql.Connection)myconn).getLargeObjectAPI();
```

where myconn is an open Connection to PostgreSQL.

Returns: LargeObject object that implements the API

Throws: SQLException by LargeObject when initializing for first time

```
public void addDataType(String type, String name)
```

This allows client code to add a handler for one of PostgreSQL's more unique data types. Normally, a data type not known by the driver is returned by ResultSet.getObject() as a PGobject instance. This method allows you to write a class that extends PGobject, and tell the driver the type name, and class name to use. The down side to this, is that you must call this method each time a connection is made.

The best way to use this is as follows:

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

where myconn is an open Connection to PostgreSQL. The handling class must extend `org.postgresql.util.PGobject`.

8.6.1.2. Class `org.postgresql.Fastpath`

```
public class Fastpath extends Object

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

`Fastpath` is an API that exists within the libpq C interface, and allows a client machine to execute a function on the database backend. Most client code will not need to use this method, but it is provided because the Large Object API uses it.

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

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

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

```
Fastpath fp = ((org.postgresql.Connection)conn).getFastpathAPI();
```

This will return an instance associated with the database connection that you can use to issue commands. The casing of `Connection` to `org.postgresql.Connection` is required, as the `getFastpathAPI()` is an extension method, not part of JDBC. Once you have a `Fastpath` instance, you can use the `fastpath()` methods to execute a backend function.

See Also: `FastpathFastpathArg`, `LargeObject`

8.6.1.2.1. Methods

```
public Object fastpath(int fnid,
                       boolean resulttype,
                       FastpathArg args[]) throws SQLException
```

Send a function call to the PostgreSQL backend.

Parameters: fnid - Function id resulttype - True if the result is an integer, false for other results args - FastpathArguments to pass to fastpath

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

```
public Object fastpath(String name,
                       boolean resulttype,
                       FastpathArg args[]) throws SQLException
```

Send a function call to the PostgreSQL backend by name.

> **Note:** The mapping for the procedure name to function id needs to exist, usually to an earlier call to addfunction(). This is the preferred method to call, as function id's can/may change between versions of the backend. For an example of how this works, refer to org.postgresql.LargeObject

Parameters: name - Function name resulttype - True if the result is an integer, false for other results args - FastpathArguments to pass to fastpath

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

See Also: `LargeObject`

```
public int getInteger(String name,
                      FastpathArg args[]) throws SQLException
```

This convenience method assumes that the return value is an Integer

Parameters: name - Function name args - Function arguments

Returns: integer result

Throws: SQLException if a database-access error occurs or no result

```
public byte[] getData(String name,
                      FastpathArg args[]) throws SQLException
```

This convenience method assumes that the return value is binary data.

Parameters: name - Function name args - Function arguments

Returns: byte[] array containing result

Throws: SQLException if a database-access error occurs or no result

```
public void addFunction(String name,
                        int fnid)
```

This adds a function to our look-up table. User code should use the `addFunctions` method, which is based upon a query, rather than hard coding the oid. The oid for a function is not guaranteed to remain static, even on different servers of the same version.

```
public void addFunctions(ResultSet rs) throws SQLException
```

This takes a `ResultSet` containing two columns. Column 1 contains the function name, Column 2 the oid. It reads the entire `ResultSet`, loading the values into the function table.

Important: Remember to `close()` the `ResultSet` after calling this!

Implementation note about function name look-ups: PostgreSQL stores the function id's and their corresponding names in the pg_proc table. To speed things up locally, instead of querying each function from that table when required, a `Hashtable` is used. Also, only the function's required are entered into this table, keeping connection times as fast as possible.

The `org.postgresql.LargeObject` class performs a query upon its start-up, and passes the returned `ResultSet` to the `addFunctions()` method here. Once this has been done, the Large Object API refers to the functions by name.

Do not think that manually converting them to the oid's will work. Okay, they will for now, but they can change during development (there was some discussion about this for V7.0), so this is implemented to prevent any unwarranted headaches in the future.

See Also: `LargeObjectManager`

```
public int getID(String name) throws SQLException
```

This returns the function id associated by its name If addFunction() or addFunctions() have not been called for this name, then an SQLException is thrown.

8.6.1.3. Class `org.postgresql.fastpath.FastpathArg`

```
public class FastpathArg extends Object

java.lang.Object
   |
   +----org.postgresql.fastpath.FastpathArg
```

Each fastpath call requires an array of arguments, the number and type dependent on the function being called. This class implements methods needed to provide this capability.

For an example on how to use this, refer to the `org.postgresql.LargeObject` package.

See Also: `Fastpath`, `LargeObjectManager`, `LargeObject`

8.6.1.3.1. Constructors

```
public FastpathArg(int value)
```

Constructs an argument that consists of an integer value

Parameters: value - int value to set

```
public FastpathArg(byte bytes[])
```

Constructs an argument that consists of an array of bytes

Parameters: bytes - array to store

```
public FastpathArg(byte buf[],
                   int off,
                   int len)
```

Constructs an argument that consists of part of a byte array

Parameters:

buf

source array

off

offset within array

len

length of data to include

```
public FastpathArg(String s)
```

Constructs an argument that consists of a String.

8.6.2. Geometric Data Types

PostgreSQL has a set of data types that can store geometric features into a table. These include single points, lines, and polygons. We support these types in Java with the org.postgresql.geometric package. It contains classes that extend the org.postgresql.util.PGobject class. Refer to that class for details on how to implement your own data type handlers.

```
Class org.postgresql.geometric.PGbox

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGbox

   public class PGbox extends PGobject implements Serializable,
Cloneable

   This represents the box data type within PostgreSQL.

Variables

 public PGpoint point[]

          These are the two corner points of the box.

Constructors

 public PGbox(double x1,
              double y1,
              double x2,
              double y2)
```

```
        Parameters:
                x1 - first x coordinate
                y1 - first y coordinate
                x2 - second x coordinate
                y2 - second y coordinate

 public PGbox(PGpoint p1,
              PGpoint p2)

        Parameters:
                p1 - first point
                p2 - second point

 public PGbox(String s) throws SQLException

        Parameters:
                s - Box definition in PostgreSQL syntax

        Throws: SQLException
                if definition is invalid

 public PGbox()

          Required constructor

Methods

 public void setValue(String value) throws SQLException

          This method sets the value of this object. It should be
overridden, but still called by subclasses.

        Parameters:
                value - a string representation of the value of the
object
        Throws: SQLException
                thrown if value is invalid for this type

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject
 public Object clone()

          This must be overridden to allow the object to be cloned

        Overrides:
                clone in class PGobject

 public String getValue()

        Returns:
                the PGbox in the syntax expected by PostgreSQL

        Overrides:
                getValue in class PGobject

Class org.postgresql.geometric.PGcircle

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGcircle

   public class PGcircle extends PGobject implements Serializable,
Cloneable

   This represents PostgreSQL's circle data type, consisting of a point
and a radius

Variables

 public PGpoint center

          This is the center point

public double radius

          This is the radius

Constructors

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

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

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

```
       Parameters:
               c - PGpoint describing the circle’s center
               r - radius of circle

public PGcircle(String s) throws SQLException

       Parameters:
               s - definition of the circle in PostgreSQL’s syntax.

       Throws: SQLException
               on conversion failure

public PGcircle()

         This constructor is used by the driver.

Methods

public void setValue(String s) throws SQLException

       Parameters:
               s - definition of the circle in PostgreSQL’s syntax.

       Throws: SQLException
               on conversion failure

       Overrides:
               setValue in class PGobject

public boolean equals(Object obj)

       Parameters:
               obj - Object to compare with

       Returns:
               true if the two 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 org.postgresql.geometric.PGline

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGline

   public class PGline extends PGobject implements Serializable,
Cloneable

   This implements a line consisting of two points. Currently line is
not yet implemented in the backend, but this class ensures that when
it’s done were ready for it.

Variables

 public PGpoint point[]

          These are the two points.

Constructors

 public PGline(double x1,
               double y1,
               double x2,
               double y2)

        Parameters:
                x1 - coordinate for first point
                y1 - coordinate for first point
                x2 - coordinate for second point
                y2 - coordinate for second point

 public PGline(PGpoint p1,
               PGpoint p2)

        Parameters:
                p1 - first point
                p2 - second point

 public PGline(String s) throws SQLException

        Parameters:
                s - definition of the 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 org.postgresql.geometric.PGlseg

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGlseg

   public class PGlseg extends PGobject implements Serializable,
Cloneable

   This implements a lseg (line segment) consisting of two points

Variables

 public PGpoint point[]

          These are the two points.

Constructors

 public PGlseg(double x1,
               double y1,
               double x2,
               double y2)

        Parameters:

                x1 - coordinate for first point
                y1 - coordinate for first point
                x2 - coordinate for second point
                y2 - coordinate for second point

 public PGlseg(PGpoint p1,
               PGpoint p2)

        Parameters:
                p1 - first point
                p2 - second point

 public PGlseg(String s) throws SQLException

        Parameters:
                s - definition of the 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 org.postgresql.geometric.PGpath

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGpath

   public class PGpath extends PGobject implements Serializable,
Cloneable

   This implements a path (a multiply segmented line, which may be
closed)

Variables

 public boolean open

          True if the path is open, false if closed

 public PGpoint points[]

          The points defining this path
```

```
Constructors

 public PGpath(PGpoint points[],
               boolean open)

        Parameters:
                points - the PGpoints that define the path
                open - True if the path is open, false if closed

 public PGpath()

          Required by the driver

 public PGpath(String s) throws SQLException

        Parameters:
                s - definition of the 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 org.postgresql.geometric.PGpoint

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGpoint

   public class PGpoint extends PGobject implements Serializable,
Cloneable

   This implements a version of java.awt.Point, except it uses double
to represent the coordinates.

   It maps to the point data type in PostgreSQL.

Variables

 public double x

          The X coordinate of the point

 public double y

          The Y coordinate of the point

Constructors

 public PGpoint(double x,
                double y)

        Parameters:
                x - coordinate
                y - coordinate

 public PGpoint(String value) throws SQLException

          This is called mainly from the other geometric types, when a
point is embedded within their definition.

        Parameters:
                value - Definition of this point in PostgreSQL’s
syntax

 public PGpoint()

          Required by the driver

Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of this point in PostgreSQL’s syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two 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

 public void translate(int x,
                       int y)

          Translate the point with the supplied amount.

        Parameters:
                x - integer amount to add on the x axis
                y - integer amount to add on the y axis

 public void translate(double x,
                       double y)

          Translate the point with the supplied amount.

        Parameters:
                x - double amount to add on the x axis
                y - double amount to add on the y axis

 public void move(int x,
                  int y)

          Moves the point to the supplied coordinates.

        Parameters:
                x - integer coordinate
                y - integer coordinate

public void move(double x,
                  double y)

          Moves the point to the supplied coordinates.

        Parameters:
                x - double coordinate
                y - double coordinate

 public void setLocation(int x,
                         int y)

          Moves the point to the supplied coordinates. refer to
          java.awt.Point for description of this

        Parameters:
                x - integer coordinate
                y - integer coordinate

        See Also:
                Point
 public void setLocation(Point p)

          Moves the point to the supplied java.awt.Point refer to
          java.awt.Point for description of this

        Parameters:
                p - Point to move to

        See Also:
                Point

Class org.postgresql.geometric.PGpolygon

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.postgresql.geometric.PGpolygon

   public class PGpolygon extends PGobject implements Serializable,
Cloneable

   This implements the polygon data type within PostgreSQL.

Variables

 public PGpoint points[]

          The points defining the polygon

Constructors

 public PGpolygon(PGpoint points[])

          Creates a polygon using an array of PGpoints

        Parameters:
                points - the points defining the polygon

 public PGpolygon(String s) throws SQLException

        Parameters:
                s - definition of the 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
```

8.6.3. Large Objects

Large objects are supported in the standard JDBC specification. However, that interface is limited, and the API provided by PostgreSQL allows for random access to the objects contents, as if it was a local file.

The org.postgresql.largeobject package provides to Java the libpq C interface’s large object API. It consists of two classes, `LargeObjectManager`, which deals with creating, opening and deleting large objects, and `LargeObject` which deals with an individual object.

8.6.3.1. Class `org.postgresql.largeobject.LargeObject`

```
public class LargeObject extends Object
```

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

This class implements the large object interface to PostgreSQL.

It provides the basic methods required to run the interface, plus a pair of methods that provide `InputStream` and `OutputStream` classes for this object.

Normally, client code would use the 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 org.postgresql.largeobject.LargeObjectManager on how to gain access to a Large Object, or how to create one.

See Also: `LargeObjectManager`

8.6.3.1.1. Variables

public static final int SEEK_SET

Indicates a seek from the beginning of a file

public static final int SEEK_CUR

Indicates a seek from the current position

public static final int SEEK_END

Indicates a seek from the end of a file

8.6.3.1.2. Methods

```
public int getOID()
```

Returns the OID of this `LargeObject`

```
public void close() throws SQLException
```

This method closes the object. You must not call methods in this object after this is called.

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

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

```
public 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

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

Writes an array to the object

```
public void write(byte buf[],
                  int off,
                  int len) throws SQLException
```

Writes some data from an array to the object

Parameters:

buf

destination array

off

offset within array

len

number of bytes to write

8.6.3.2. Class `org.postgresql.largeobject.LargeObjectManager`

```
public class LargeObjectManager extends Object

java.lang.Object
   |
   +----org.postgresql.largeobject.LargeObjectManager
```

This class implements the large object interface to PostgreSQL. It provides methods that allow client code to create, open and delete large objects from the database. When opening an object, an instance of `org.postgresql.largeobject.LargeObject` is returned, and its methods then allow access to the object.

This class can only be created by org.postgresql.Connection. To get access to this class, use the following segment of code:

```
import org.postgresql.largeobject.*;
Connection  conn;
LargeObjectManager lobj;
// ... code that opens a connection ...
lobj = ((org.postgresql.Connection)myconn).getLargeObjectAPI();
```

Normally, client code would use the 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 org.postgresql.largeobject.LargeObject on how to manipulate the contents of a Large Object.

8.6.3.2.1. Variables

public static final int WRITE

This mode indicates we want to write to an object.

public static final int READ

This mode indicates we want to read an object.

public static final int READWRITE

This mode is the default. It indicates we want read and write access to a large object.

8.6.3.2.2. Methods

```
public LargeObject open(int oid) throws SQLException
```

This opens an existing large object, based on its OID. This method assumes that READ and WRITE access is required (the default).

```
public LargeObject open(int oid,
                        int mode) throws SQLException
```

This opens an existing large object, based on its OID, and allows setting the access mode.

```
public int create() throws SQLException
```

This creates a large object, returning its OID. It defaults to READWRITE for the new object's attributes.

```
public int create(int mode) throws SQLException
```

This creates a large object, returning its OID, and sets the access mode.

```
public void delete(int oid) throws SQLException
```

This deletes a large object.

```
public void unlink(int oid) throws SQLException
```

This deletes a large object. It is identical to the delete method, and is supplied as the C API uses unlink.

8.6.4. Object Serialization

PostgreSQL is not a normal SQL database. It is 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)
```

Okay, 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 Large Object to store them. However, you cannot perform queries on those objects.

What the org.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 org.postgresql.util.Serialize

java.lang.Object
   |
   +----org.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 org.postgresql.util package contains classes used by the internals of
the main driver, and the other extensions.

Class org.postgresql.util.PGmoney

java.lang.Object
   |
   +----org.postgresql.util.PGobject
           |
           +----org.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 imbeded 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 org.postgresql.util.PGobject

java.lang.Object
   |
   +----org.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 org.postgresql.Connection permits a class that extends this
```

```
class to be associated with a named type. This is how the
org.postgresql.geometric package operates.
    ResultSet.getObject() will return this class for any type that is
not recognized on having it’s own handler. Because of this, any
PostgreSQL data type is supported.

Constructors

 public PGobject()

          This is called by org.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 org.postgresql.util.PGtokenizer

java.lang.Object
   |
   +----org.postgresql.util.PGtokenizer

   public class PGtokenizer extends Object

   This class is used to tokenize the text output of PostgreSQL.

   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 tokenizer.

        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 data types 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 org.postgresql.util.Serialize

This was documented earlier under Object Serialization.

Class org.postgresql.util.UnixCrypt

java.lang.Object
   |
   +----org.postgresql.util.UnixCrypt

   public class UnixCrypt extends Object

   This class provides us with the ability to encrypt passwords when
sent over the network stream

   Contains static methods to encrypt and compare passwords with Unix
encrypted passwords.

   See John Dumas’s Java Crypt page for the original source.

   http://www.zeh.com/local/jfd/crypt.html

Methods

 public static final String crypt(String salt,
                                  String original)

          Encrypt a password given the clear-text password and a
"salt".

        Parameters:
                salt - A two-character string representing the salt
used
                to iterate the encryption engine in lots of different
                ways. If you are generating a new encryption then this
                value should be randomized.
                original - The password to be encrypted.
        Returns:
                A string consisting of the 2-character salt followed
by
                the encrypted password.

 public static final String crypt(String original)

          Encrypt a password given the clear-text password. This method
generates a random salt using the 'java.util.Random' class.

        Parameters:
                original - The password to be encrypted.

        Returns:
                A string consisting of the 2-character salt followed
by
                the encrypted password.

 public static final boolean matches(String encryptedPassword,
                                     String enteredPassword)

          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 acquired).

        Returns:
                true if the password should be considered correct.
```

8.7. Using the driver in a multi-threaded or a servlet environment

A problem with many JDBC drivers is that only one thread can use a `Connection` at any one time -- otherwise a thread could send a query while another one is receiving results, and this would be a bad thing for the database engine.

PostgreSQL 6.4 brought thread safety to the entire driver. (Standard JDBC was thread safe in 6.3, but the Fastpath API was not.) Consequently, if your application uses multiple threads then you do not have to worry about complex algorithms to ensure that only one uses the database at any time.

If a thread attempts to use the connection while another one is using it, it will wait until the other thread has finished its current operation. If it is a regular SQL statement, then the operation consists of sending the statement and retrieving any `ResultSet` (in full). If it is a `Fastpath` call (e.g., reading a block from a `LargeObject`) then it is the time to send and retrieve that block.

This is fine for applications and applets but can cause a performance problem with servlets. With servlets you can have a heavy load on the connection. If you have several threads performing queries then each but one will pause, which may not be what you are after.

To solve this, you would be advised to create a pool of connections. When ever a thread needs to use the database, it asks a manager class for a `Connection`. The manager hands a free connection to the thread and marks it as busy. If a free connection is not available, it opens one. Once the thread has finished with it, it returns it to the manager who can then either close it or add it to the pool. The manager would also check that the connection is still alive and remove it from the pool if it is dead.

So, with servlets, it is up to you to use either a single connection, or a pool. The plus side for a pool is that threads will not be hit by the bottle neck caused by a single network connection. The down side is that it increases the load on the server, as a backend process is created for each `Connection`. It is up to you and your applications requirements.

8.8. Further Reading

If you have not yet read it, I'd advise you read the JDBC API Documentation (supplied with Sun's JDK), and the JDBC Specification. Both are available from
http://java.sun.com/products/jdbc/index.html.

http://jdbc.postgresql.org contains updated information not included in this document, and also includes precompiled drivers.

Chapter 9. PyGreSQL - Python Interface

Author: Written by D'Arcy J.M. Cain (<darcy@druid.net>). Based heavily on code written by Pascal Andre <andre@chimay.via.ecp.fr>. Copyright © 1995, Pascal Andre. Further modifications Copyright © 1997-2000 by D'Arcy J.M. Cain.

9.1. The pg Module

You may either choose to use the old mature interface provided by the `pg` module or otherwise the newer `pgdb` interface compliant with the DB-API 2.0 (http://www.python.org/topics/database/DatabaseAPI-2.0.html) specification developed by the Python DB-SIG.

Here we describe only the older `pg` API. As long as PyGreSQL does not contain a description of the DB-API you should read about the API at http://www.python.org/topics/database/DatabaseAPI-2.0.html.

A tutorial-like introduction to the DB-API can be found at http://www2.linuxjournal.com/lj-issues/issue49/2605.html

The `pg` module defines three objects:

- `pgobject`, which handles the connection and all the requests to the database,
- `pglargeobject`, which handles all the accesses to Postgres large objects, and
- `pgqueryobject` that handles query results.

If you want to see a simple example of the use of some of these functions, see http://www.druid.net/rides where I have a link at the bottom to the actual Python code for the page.

9.1.1. Constants

Some constants are defined in the `pg` module dictionary. They are intended to be used as a parameters for methods calls. You should refer to the `libpq` description (Chapter 1) for more information about them. These constants are:

`INV_READ`
`INV_WRITE`
`INV_ARCHIVE`

large objects access modes, used by `(pgobject.)locreate` and `(pglarge.)open`.

`SEEK_SET`
`SEEK_CUR`
`SEEK_END`

positional flags, used by `(pglarge.)seek`.

`version`
`__version__`

constants that give the current version

9.2. `pg` Module Functions

`pg` module defines only a few methods that allow to connect to a database and to define default variables that override the environment variables used by PostgreSQL.

These default variables were designed to allow you to handle general connection parameters without heavy code in your programs. You can prompt the user for a value, put it in the default variable, and forget it, without having to modify your environment. The support for default variables can be disabled by setting the -DNO_DEF_VAR option in the Python Setup file. Methods relative to this are specified by te tag [DV].

All variables are set to None at module initialization, specifying that standard environment variables should be used.

connect

Name

`connect` opens a connection to the database server

Synopsis

```
connect([dbname], [host], [port], [opt], [tty], [user], [passwd])
```

Parameters

dbname

Name of connected database (string/None).

host

Name of the server host (string/None).

port

Port used by the database server (integer/-1).

opt

Options for the server (string/None).

tty

File or tty for optional debug output from backend (string/None).

user

PostgreSQL user (string/None).

passwd

Password for user (string/None).

Return Type

pgobject

If successful, an object handling a database connection is returned.

Exceptions

TypeError

Bad argument type, or too many arguments.

SyntaxError

Duplicate argument definition.

pg.error

Some error occurred during pg connection definition.

(+ all exceptions relative to object allocation)

Description

This method opens a connection to a specified database on a given PostgreSQL server. You can use keywords here, as described in the Python tutorial. The names of the keywords are the name of the parameters given in the syntax line. For a precise description of the parameters, please refer to the PostgreSQL user manual.

Examples

```
import pg

con1 = pg.connect('testdb', 'myhost', 5432, None, None, 'bob', None)
con2 = pg.connect(dbname='testdb', host='localhost', user='bob')
```

get_defhost

Name

`get_defhost` get default host name [DV]

Synopsis

```
get_defhost()
```

Parameters

none

Return Type

string or None

Default host specification

Exceptions

SyntaxError

Too many arguments.

Description

`get_defhost()` returns the current default host specification, or None if the environment variables should be used. Environment variables will not be looked up.

set_defhost

Name

`set_defhost` set default host name [DV]

Synopsis

```
set_defhost(host)
```

Parameters

`host`

New default host (string/None).

Return Type

string or None

Previous default host specification.

Exceptions

TypeError

Bad argument type, or too many arguments.

Description

`set_defhost()` sets the default host value for new connections. If None is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default host.

get_defport

Name

`get_defport` get default port [DV]

Synopsis

```
get_defport()
```

Parameters

none

Return Type

integer or None

Default port specification

Exceptions

SyntaxError

Too many arguments.

Description

`get_defport()` returns the current default port specification, or None if the environment variables should be used. Environment variables will not be looked up.

set_defport

Name

`set_defport` set default port [DV]

Synopsis

`set_defport(*port*)`

Parameters

`port`

New default host (integer/-1).

Return Type

integer or None

Previous default port specification.

Exceptions

TypeError

Bad argument type, or too many arguments.

Description

`set_defport()` sets the default port value for new connections. If -1 is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default port.

get_defopt

Name

`get_defopt` get default options specification [DV]

Synopsis

`get_defopt()`

Parameters

none

Return Type

string or None

Default options specification

Exceptions

SyntaxError

Too many arguments.

Description

`get_defopt()` returns the current default connection options specification, or None if the environment variables should be used. Environment variables will not be looked up.

set_defopt

Name

`set_defopt` set options specification [DV]

Synopsis

`set_defopt(`*`options`*`)`

Parameters

`options`

New default connection options (string/None).

Return Type

string or None

Previous default opt specification.

Exceptions

TypeError

Bad argument type, or too many arguments.

Description

`set_defopt()` sets the default connection options value for new connections. If None is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default options.

get_deftty

Name

`get_deftty` get default connection debug terminal specification [DV]

Synopsis

```
get_deftty()
```

Parameters

none

Return Type

string or None

Default debug terminal specification

Exceptions

SyntaxError

Too many arguments.

Description

`get_deftty()` returns the current default debug terminal specification, or None if the environment variables should be used. Environment variables will not be looked up.

set_deftty

Name

`set_deftty` set default debug terminal specification [DV]

Synopsis

`set_deftty(`*`terminal`*`)`

Parameters

`terminal`

New default debug terminal (string/None).

Return Type

string or None

Previous default debug terminal specification.

Exceptions

TypeError

Bad argument type, or too many arguments.

Description

`set_deftty()` sets the default terminal value for new connections. If None is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default terminal.

get_defbase

Name

`get_defbase` get default database name specification [DV]

Synopsis

```
get_defbase()
```

Parameters

none

Return Type

string or None

Default debug database name specification

Exceptions

SyntaxError

Too many arguments.

Description

`get_defbase()` returns the current default database name specification, or None if the environment variables should be used. Environment variables will not be looked up.

set_defbase

Name

`set_defbase` set default database name specification [DV]

Synopsis

`set_defbase(`*`database`*`)`

Parameters

`database`

New default database name (string/None).

Return Type

string or None

Previous default database name specification.

Exceptions

TypeError

Bad argument type, or too many arguments.

Description

`set_defbase()` sets the default database name for new connections. If None is supplied as parameter, environment variables will be used in future connections. It returns the previous setting for default database name.

9.3. Connection object: `pgobject`

This object handles a connection to the PostgreSQL database. It embeds and hides all the parameters that define this connection, leaving just really significant parameters in function calls.

Some methods give direct access to the connection socket. They are specified by the tag [DA]. *Do not use them unless you really know what you are doing.* If you prefer disabling them, set the `-DNO_DIRECT` option in the Python `Setup` file.

Some other methods give access to large objects. if you want to forbid access to these from the module, set the `-DNO_LARGE` option in the Python `Setup` file. These methods are specified by the tag [LO].

Every `pgobject` defines a set of read-only attributes that describe the connection and its status. These attributes are:

host

the host name of the server (string)

port

the port of the server (integer)

db

the selected database (string)

options

the connection options (string)

tty

the connection debug terminal (string)

user

user name on the database system (string)

status

the status of the connection (integer: 1 - OK, 0 - BAD)

error

the last warning/error message from the server (string)

query

Name

`query` executes a SQL command

Synopsis

`query(`*`command`*`)`

Parameters

`command`

SQL command (string).

Return Type

pgqueryobject or None

Result values.

Exceptions

TypeError

Bad argument type, or too many arguments.

ValueError

Empty SQL query.

pg.error

Error during query processing, or invalid connection.

Description

`query()` method sends a SQL query to the database. If the query is an insert statement, the return value is the OID of the newly inserted row. If it is otherwise a query that does not return a result (i.e., is not a some kind of `SELECT` statement), it returns None. Otherwise, it returns a `pgqueryobject` that can be accessed via the `getresult()` or `dictresult()` methods or simply printed.

reset

Name

`reset` resets the connection

Synopsis

```
reset()
```

Parameters

none

Return Type

none

Exceptions

TypeError

Too many (any) arguments.

Description

`reset()` method resets the current database.

close

Name

`close` close the database connection

Synopsis

```
close()
```

Parameters

none

Return Type

none

Exceptions

TypeError

Too many (any) arguments.

Description

`close()` method closes the database connection. The connection will be closed in any case when the connection is deleted but this allows you to explicitly close it. It is mainly here to allow the DB-SIG API wrapper to implement a close function.

fileno

Name

`fileno` returns the socket used to connect to the database

Synopsis

```
fileno()
```

Parameters

none

Return Type

socket id

The underlying socket id used to connect to the database.

Exceptions

TypeError

Too many (any) arguments.

Description

`fileno()` method returns the underlying socket id used to connect to the database. This is useful for use in `select` calls, etc.

getnotify

Name

`getnotify` gets the last notify from the server

Synopsis

```
getnotify()
```

Parameters

none

Return Type

tuple, None

Last notify from server

Exceptions

TypeError

Too many (any) arguments.

pg.error

Invalid connection.

Description

`getnotify()` method tries to get a notify from the server (from the `SQL` statement `NOTIFY`). If the server returns no notify, the methods returns None. Otherwise, it returns a tuple (couple) `(relname, pid)`, where `relname` is the name of the notify and `pid` the process id of the connection that triggered the notify. Remember to do a listen query first otherwise getnotify will always return None.

inserttable

Name

`inserttable` inserts a list into a table

Synopsis

`inserttable(`*`table`*`, `*`values`*`)`

Parameters

`table`

The table name (string).

`values`

The list of rows values to insert (list).

Return Type

none

Exceptions

TypeError

Bad argument type or too many (any) arguments.

pg.error

Invalid connection.

Description

`inserttable()` method allows to quickly insert large blocks of data in a table: it inserts the whole values list into the given table. The list is a list of tuples/lists that define the values for each inserted row. The rows values may contain string, integer, long or double (real) values. *Be very careful:* this method does not typecheck the fields according to the table definition; it just look whether or not it knows how to handle such types.

putline

Name

`putline` writes a line to the server socket [DA]

Synopsis

`putline(`*`line`*`)`

Parameters

`line`

Line to be written (string).

Return Type

none

Exceptions

TypeError

Bad argument type or too many (any) arguments.

pg.error

Invalid connection.

Description

`putline()` method allows to directly write a string to the server socket.

getline

Name

`getline` gets a line from server socket [DA]

Synopsis

```
getline()
```

Parameters

none

Return Type

string

The line read.

Exceptions

TypeError

Bad argument type or too many (any) arguments.

pg.error

Invalid connection.

Description

`getline()` method allows to directly read a string from the server socket.

endcopy

Name

`endcopy` synchronizes client and server [DA]

Synopsis

```
endcopy()
```

Parameters

none

Return Type

none

Exceptions

TypeError

  Bad argument type or too many (any) arguments.

pg.error

  Invalid connection.

Description

The use of direct access methods may desynchronize client and server. This method ensure that client and server will be synchronized.

locreate

Name

`locreate` creates of large object in the database [LO]

Synopsis

`locreate(`*`mode`*`)`

Parameters

`mode`

Large object create mode.

Return Type

pglarge

Object handling the PostgreSQL large object.

Exceptions

TypeError

Bad argument type or too many arguments.

pg.error

Invalid connection, or creation error.

Description

`locreate()` method creates a large object in the database. The mode can be defined by OR-ing the constants defined in the pg module (`INV_READ, INV_WRITE` and `INV_ARCHIVE`).

getlo

Name

`getlo` builds a large object from given `oid` [LO]

Synopsis

`getlo(`*`oid`*`)`

Parameters

`oid`

`OID` of the existing large object (integer).

Return Type

pglarge

Object handling the PostgreSQL large object.

Exceptions

TypeError

Bad argument type or too many arguments.

pg.error

Invalid connection.

Description

`getlo()` method allows to reuse a formerly created large object through the `pglarge` interface, providing the user have its `oid`.

loimport

Name

`loimport` imports a file to a PostgreSQL large object [LO]

Synopsis

`loimport(`*`filename`*`)`

Parameters

`filename`

The name of the file to be imported (string).

Return Type

pglarge

Object handling the PostgreSQL large object.

Exceptions

TypeError

Bad argument type or too many arguments.

pg.error

Invalid connection, or error during file import.

Description

`loimport()` method allows to create large objects in a very simple way. You just give the name of a file containing the data to be use.

9.4. Database wrapper class: `DB`

`pg` module contains a class called `DB`. All `pgobject` methods are included in this class also. A number of additional `DB` class methods are described below. The preferred way to use this module is as follows (See description of the initialization method below.):

```
import pg

db = pg.DB(...)

for r in db.query(
    "SELECT foo,bar
       FROM foo_bar_table
```

```
      WHERE foo !~ bar"
    ).dictresult():

    print '%(foo)s %(bar)s' % r
```

The following describes the methods and variables of this class.

The `DB` class is initialized with the same arguments as the `pg.connect` method. It also initializes a few internal variables. The statement `db = DB()` will open the local database with the name of the user just like `pg.connect()` does.

pkey

Name

`pkey` returns the primary key of a table

Synopsis

`pkey(`*`table`*`)`

Parameters

`table`

name of table.

Return Type

string

Name of field which is the primary key of the table.

Description

`pkey()` method returns the primary key of a table. Note that this raises an exception if the table does not have a primary key.

get_databases

Name

`get_databases` get list of databases in the system

Synopsis

```
get_databases()
```

Parameters

none

Return Type

list

List of databases in the system.

Description

Although you can do this with a simple select, it is added here for convenience

get_tables

Name

`get_tables` get list of tables in connected database

Synopsis

```
get_tables()
```

Parameters

none

Return Type

list

List of tables in connected database.

Description

Although you can do this with a simple select, it is added here for convenience

get_attnames

Name

`get_attnames` returns the attribute names of a table

Synopsis

`get_attnames(`*`table`*`)`

Parameters

`table`

name of table.

Return Type

list

List of attribute names.

Description

Given the name of a table, digs out the list of attribute names.

get

Name

`get` get a tuple from a database table

Synopsis

get(*table*, *arg*, [*keyname*])

Parameters

table

Name of table.

arg

Either a dictionary or the value to be looked up.

[*keyname*]

Name of field to use as key (optional).

Return Type

dictionary

A dictionary mapping attribute names to row values.

Description

This method is the basic mechanism to get a single row. It assumes that the key specifies a unique row. If keyname is not specified then the primary key for the table is used. If arg is a dictionary then the value for the key is taken from it and it is modified to include the new values, replacing existing values where necessary. The oid is also put into the dictionary but in order to allow the caller to work with multiple tables, the attribute name is munged to make it unique. It consists of the string `oid_` followed by the name of the table.

insert

Name

`insert` insert a tuple into a database table

Synopsis

`insert(`*`table`*`, `*`a`*`)`

Parameters

`table`

Name of table.

`a`

A dictionary of values.

Return Type

integer

The OID of the newly inserted row.

Description

This method inserts values into the table specified filling in the values from the dictionary. It then reloads the dictionary with the values from the database. This causes the dictionary to be updated with values that are modified by rules, triggers, etc.

update

Name

`update` update a database table

Synopsis

`update(`*`table`*`, `*`a`*`)`

Parameters

`table`

Name of table.

`a`

A dictionary of values.

Return Type

integer

The OID of the newly updated row.

Description

Similar to insert but updates an existing row. The update is based on the OID value as munged by get. The array returned is the one sent modified to reflect any changes caused by the update due to triggers, rules, defaults, etc.

clear

Name

`clear` clear a database table

Synopsis

`clear(`*`table`*`, [`*`a`*`])`

Parameters

`table`

Name of table.

[*`a`*]

A dictionary of values.

Return Type

dictionary

A dictionary with an empty row.

Description

This method clears all the attributes to values determined by the types. Numeric types are set to 0, dates are set to `'today'` and everything else is set to the empty string. If the array argument is present, it is used as the array and any entries matching attribute names are cleared with everything else left unchanged.

delete

Name

`delete` deletes the row from a table

Synopsis

`delete(table, [a])`

Parameters

`table`

Name of table.

[*`a`*]

A dictionary of values.

Return Type

none

Description

This method deletes the row from a table. It deletes based on the OID as munged as described above.

9.5. Query result object: `pgqueryobject`

getresult

Name

`getresult` gets the values returned by the query

Synopsis

```
getresult()
```

Parameters

none

Return Type

list

List of tuples.

Exceptions

SyntaxError

Too many arguments.

pg.error

Invalid previous result.

Description

`getresult()` method returns the list of the values returned by the query. More information about this result may be accessed using `listfields`, `fieldname` and `fieldnum` methods.

dictresult

Name

`dictresult` like getresult but returns a list of dictionaries

Synopsis

```
dictresult()
```

Parameters

none

Return Type

list

List of dictionaries.

Exceptions

SyntaxError

Too many arguments.

pg.error

Invalid previous result.

Description

`dictresult()` method returns the list of the values returned by the query with each tuple returned as a dictionary with the field names used as the dictionary index.

listfields

Name

`listfields` lists the fields names of the query result

Synopsis

```
listfields()
```

Parameters

none

Return Type

list

field names

Exceptions

SyntaxError

Too many arguments.

pg.error

Invalid query result, or invalid connection.

Description

`listfields()` method returns the list of field names defined for the query result. The fields are in the same order as the result values.

fieldname

Name

`fieldname` field number-name conversion

Synopsis

`fieldname(i)`

Parameters

`i`

field number (integer).

Return Type

string

field name.

Exceptions

TypeError

Bad parameter type, or too many arguments.

`ValueError`

Invalid field number.

pg.error

Invalid query result, or invalid connection.

Description

`fieldname()` method allows to find a field name from its rank number. It can be useful for displaying a result. The fields are in the same order than the result values.

fieldnum

Name

`fieldnum` field name-number conversion

Synopsis

`fieldnum(`*`name`*`)`

Parameters

`name`

field name (string).

Return Type

integer

field number (integer).

Exceptions

TypeError

Bad parameter type, or too many arguments.

`ValueError`

Unknown field name.

pg.error

Invalid query result, or invalid connection.

Description

`fieldnum()` method returns a field number from its name. It can be used to build a function that converts result list strings to their correct type, using a hardcoded table definition. The number returned is the field rank in the result values list.

ntuples

Name

`ntuples` returns the number of tuples in query object

Synopsis

```
ntuples()
```

Parameters

none

Return Type

integer

The number of tuples in query object.

Exceptions

SyntaxError

Too many arguments.

Description

`ntuples()` method returns the number of tuples found in a query.

9.6. Large Object: `pglarge`

This object handles all the request concerning a PostgreSQL large object. It embeds and hides all the recurrent variables (object oid and connection), exactly in the same way `pgobjects` do, thus only keeping significant parameters in function calls. It keeps a reference to the `pgobject` used for its creation, sending requests though with its parameters. Any modification but dereferencing the `pgobject` will thus affect the `pglarge` object. Dereferencing the initial `pgobject` is not a problem since Python will not deallocate it before the large object dereference it. All functions return a generic error message on call error, whatever the exact error was. The `error` attribute of the object allows to get the exact error message.

`pglarge` objects define a read-only set of attributes that allow to get some information about it. These attributes are:

oid

the oid associated with the object

pgcnx

the `pgobject` associated with the object

error

the last warning/error message of the connection

Be careful: In multithreaded environments, `error` may be modified by another thread using the same `pgobject`. Remember these object are shared, not duplicated; you should provide some locking to be able if you want to check this. The oid attribute is very interesting because it allow you reuse the oid later, creating the `pglarge` object with a `pgobject` `getlo()` method call.

See also Chapter 2 for more information about the PostgreSQL large object interface.

open

Name

`open` opens a large object

Synopsis

`open(mode)`

Parameters

mode

open mode definition (integer).

Return Type

none

Exceptions

TypeError

Bad parameter type, or too many arguments.

IOError

Already opened object, or open error.

pg.error

Invalid connection.

Description

`open()` method opens a large object for reading/writing, in the same way than the UNIX `open()` function. The mode value can be obtained by OR-ing the constants defined in the pg module (`INV_READ, INV_WRITE`).

close

Name

`close` closes the large object

Synopsis

```
close()
```

Parameters

none

Return Type

none

Exceptions

SyntaxError

Too many arguments.

IOError

Object is not opened, or close error.

pg.error

Invalid connection.

Description

`close()` method closes previously opened large object, in the same way than the UNIX `close()` function.

read

Name

`read` reads from the large object

Synopsis

read(*size*)

Parameters

size

Maximal size of the buffer to be read (integer).

Return Type

string

The read buffer.

Exceptions

TypeError

Bad parameter type, or too many arguments.

IOError

Object is not opened, or read error.

pg.error

Invalid connection or invalid object.

Description

`read()` method allows to read data from the large object, starting at current position.

write

Name

`write` writes to the large object

Synopsis

`write(`*`string`*`)`

Parameters

`string`

Buffer to be written (string).

Return Type

none

Exceptions

TypeError

Bad parameter type, or too many arguments.

IOError

Object is not opened, or write error.

pg.error

Invalid connection or invalid object.

Description

`write()` method allows to write data to the large object, starting at current position.

seek

Name

`seek` change current position in the large object

Synopsis

`seek(`*`offset`*`, `*`whence`*`)`

Parameters

`offset`

Position offset (integer).

`whence`

Positional parameter (integer).

Return Type

integer

New current position in the object.

Exceptions

TypeError

Bad parameter type, or too many arguments.

IOError

Object is not opened, or seek error.

pg.error

Invalid connection or invalid object.

Description

`seek()` method allows to move the cursor position in the large object. The whence parameter can be obtained by OR-ing the constants defined in the `pg` module (`SEEK_SET, SEEK_CUR, SEEK_END`).

tell

Name

`tell` returns current position in the large object

Synopsis

```
tell()
```

Parameters

none

Return Type

integer

Current position in the object.

Exceptions

SyntaxError

Too many arguments.

IOError

Object is not opened, or seek error.

pg.error

Invalid connection or invalid object.

Description

`tell()` method allows to get the current position in the large object.

unlink

Name

`unlink` deletes the large object

Synopsis

```
unlink()
```

Parameters

none

Return Type

none

Exceptions

SyntaxError

Too many arguments.

IOError

Object is not closed, or unlink error.

pg.error

Invalid connection or invalid object.

Description

`unlink()` method unlinks (deletes) the large object.

size

Name

`size` gives the large object size

Synopsis

```
size()
```

Parameters

none

Return Type

integer

The large object size.

Exceptions

SyntaxError

Too many arguments.

IOError

Object is not opened, or seek/tell error.

pg.error

Invalid connection or invalid object.

Description

`size()` method allows to get the size of the large object. It was implemented because this function is very useful for a WWW interfaced database. Currently the large object needs to be opened.

export

Name

`export` saves the large object to file

Synopsis

`export(`*`filename`*`)`

Parameters

`filename`

The file to be created.

Return Type

none

Exceptions

TypeError

Bad argument type, or too many arguments.

IOError

Object is not closed, or export error.

pg.error

Invalid connection or invalid object.

Description

`export()` method allows to dump the content of a large object in a very simple way. The exported file is created on the host of the program, not the server host.

9.7. DB-API Interface

* *This section needs to be written.*

See http://www.python.org/topics/database/DatabaseAPI-2.0.html for a description of the DB-API 2.0.

Chapter 10. Lisp Programming Interface

`pg.el` is a socket-level interface to Postgres for emacs.

Author: Written by Eric Marsden <`emarsden@mail.dotcom.fr`> on 1999-07-21

`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)))
```

II. Server Programming

Chapter 11. Architecture

11.1. 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):

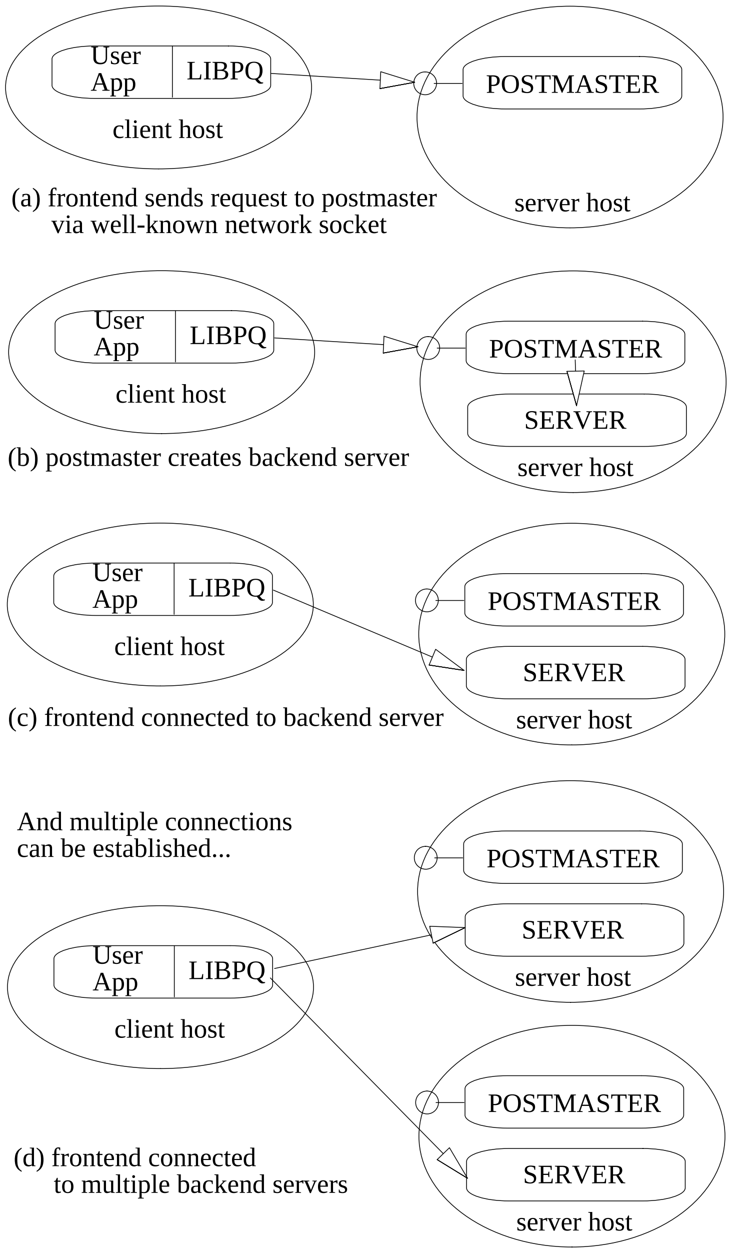
A supervisory daemon process (postmaster),

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 a cluster (of databases). Frontend applications that wish to access a given database within a cluster make calls to the library. The library sends user requests over the network to the postmaster (Figure 11-1(a)), which in turn starts a new backend server process (Figure 11-1(b))

Figure 11-1. How a connection is established


User App
LIBPQ
client host
POSTMASTER
server host
(a) frontend sends request to postmaster via well-known network socket
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(b) postmaster creates backend server
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(c) frontend connected to backend server
And multiple connections can be established...
POSTMASTER
SERVER
server host
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(d) frontend connected to multiple backend servers

and connects the frontend process to the new server (Figure 11-1(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 12. 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

12.1. 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 tables 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 tables can be modified by the user, and since Postgres bases its internal operation on these tables, 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.

12.2. 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 table. EMP is an example of a composite type.

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

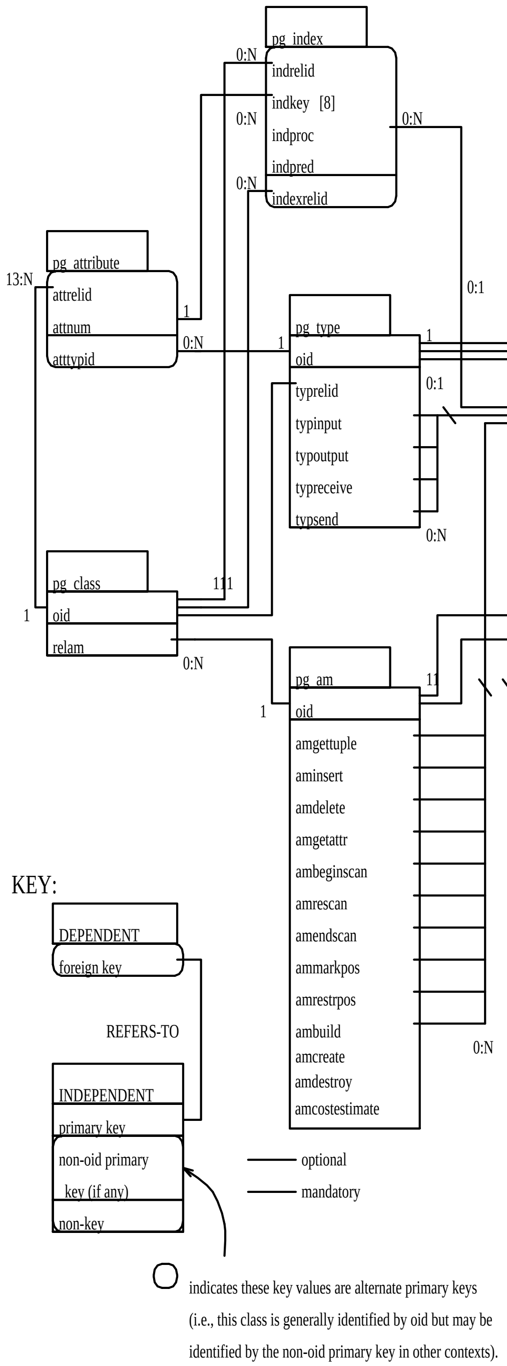
12.3. About the Postgres System Catalogs

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 tables contain information that may be useful to the end user. (There are many other system catalogs, but there should rarely be a reason to query them directly.)

Table 12-1. Postgres System Catalogs

Catalog Name	Description
pg_database	databases
pg_class	tables
pg_attribute	table columns
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 12-1. The major Postgres system catalogs



The Reference Manual gives a more detailed explanation of these catalogs and their columns. However, Figure 12-1 shows the major entities and their relationships in the system catalogs. (Columns that do not refer to other entities are not shown unless they are part of a primary key.) This diagram is more or less incomprehensible until you actually start looking at the contents of the catalogs and see how they relate to each other. For now, the main things to take away from this diagram are as follows:

In several of the sections that follow, we will present various join queries on the system catalogs that display information we need to extend the system. Looking at this diagram should make some of these join queries (which are often three- or four-way joins) more understandable, because you will be able to see that the columns used in the queries form foreign keys in other tables.

Many different features (tables, columns, functions, types, access methods, etc.) are tightly integrated in this schema. A simple create command may modify many of these catalogs.

Types and procedures are central to the schema.

Note: We use the words *procedure* and *function* more or less interchangably.

Nearly every catalog contains some reference to rows in one or both of these tables. For example, Postgres frequently uses type signatures (e.g., of functions and operators) to identify unique rows of other catalogs.

There are many columns and relationships that have obvious meanings, but there are many (particularly those that have to do with access methods) that do not. The relationships between pg_am, pg_amop, pg_amproc, pg_operator and pg_opclass are particularly hard to understand and will be described in depth (in the section on interfacing types and operators to indices) after we have discussed basic extensions.

Chapter 13. 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`.

13.1. 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 semicolons and bracketed within single-quote marks. Note that quote marks used in the queries must be escaped, by preceding them with a backslash.

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.

13.1.1. 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 1;'
LANGUAGE 'sql';
```

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

```
SELECT tp1( 17,100.0);
```

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

```
CREATE FUNCTION hobbies (EMP) RETURNS SETOF hobbies
    AS ’SELECT hobbies.* FROM hobbies
        WHERE $1.name = hobbies.person’
    LANGUAGE ’sql’;
```

13.1.2. 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 column name for the function’s result (with the name RESULT), but this column name is not visible outside the function. 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      |
+-------+
```

13.1.3. 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 ~= point ’(2,1)’;


+-----+-------+
|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 notations attribute(table) and table.attribute interchangably:

```
--
-- 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 row. We do this by assembling the entire row within the function, attribute by attribute. This is an example of a function that returns a single EMP row:

```
CREATE FUNCTION new_emp()
    RETURNS EMP
    AS 'SELECT text ''None'' AS name,
        1000 AS salary,
        25 AS age,
        point ''(2,2)'' AS cubicle'
    LANGUAGE 'sql';
```

In this case we have specified each of the attributes with a constant value, but any computation or expression could have been substituted for these constants. 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 that defined the composite type.

You must typecast the expressions to match the composite type's definition, or you will get errors like this:

```
    ERROR:  function declared to return emp returns varchar instead of text at column
1
```

When calling a function that returns a row, we cannot retrieve the entire row. We must either project an attribute out of the row or pass the entire row 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;
NOTICE: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 |
+--+
```

13.2. 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 currently three procedural languages available in the standard Postgres distribution (PLSQL, PLTCL and PLPERL), and other languages can be defined. Refer to Chapter 23 for more information.

13.3. Internal Functions

Internal functions are functions written in C that 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.

Internal functions are declared in **CREATE FUNCTION** with language name `internal`.

13.4. Compiled (C) Language Functions

Functions written in C can be compiled into dynamically loadable objects (also called shared libraries), and used to implement user-defined SQL functions. The first time a user-defined function in a particular loadable object file is called in a backend session, the dynamic loader loads that object file into memory so that the function can be called. The **CREATE FUNCTION** for a user-defined function must therefore specify two pieces of information for the function: the name of the loadable object file, and the

C name (link symbol) of the specific function to call within that object file. If the C name is not explicitly specified then it is assumed to be the same as the SQL function name.

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

The string that specifies the object file (the first string in the AS clause) should be the *full path* of the object code file for the function, bracketed by single quote marks. If a link symbol is given in the AS clause, the link symbol should also be bracketed by single quote 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.

> **Note:** 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.

Two different calling conventions are currently used for C functions. The newer "version 1" calling convention is indicated by writing a `PG_FUNCTION_INFO_V1()` macro call for the function, as illustrated below. Lack of such a macro indicates an old-style ("version 0") function. The language name specified in CREATE FUNCTION is ’C’ in either case. Old-style functions are now deprecated because of portability problems and lack of functionality, but they are still supported for compatibility reasons.

13.4.1. Base Types in C-Language Functions

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. Note that you should always include `postgres.h` first, and that in turn includes `c.h`.

Table 13-1. 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 or int16	include/postgres.h
int2vector	(int2vector *)	include/postgres.h

Built-In Type	C Type	Defined In
int4	int4 or int32	include/postgres.h
float4	(float4 *)	include/c.h or include/postgres.h
float8	(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
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 (also 8 bytes, if sizeof(Datum) is 8 on your machine). You should be careful to define your types such that they will be the same size (in bytes) on all architectures. For example, the `long` type is dangerous because it is 4 bytes on some machines and 8 bytes on others, whereas `int` type is 4 bytes on most Unix machines (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. To return a value of such a type, allocate the right amount of memory with `palloc()`, fill in the allocated memory, and return a pointer to it. (Alternatively, you can return an input value of the same type by returning its pointer. *Never* modify the contents of a pass-by-reference input value, however.)

Finally, all variable-length types must also be passed by reference. All variable-length types must begin with a length field of exactly 4 bytes, and all data to be stored within that type must be located in the memory immediately following that length field. The length field is the total length of the structure (i.e., it includes the size of the length field itself). We can define the text type as follows:

```
typedef struct {
    int4 length;
    char data[1];
} text;
```

Obviously, the data field shown here 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.

13.4.2. Version-0 Calling Conventions for C-Language Functions

We present the old style calling convention first --- although this approach is now deprecated, it's easier to get a handle on initially. In the version-0 method, the arguments and result of the C function are just declared in normal C style, but being careful to use the C representation of each SQL data type as shown above.

Here are some examples:

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

/* By Value */

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

/* By Reference, Fixed Length */

float8 *
add_one_float8(float8 *arg)
{
    float8    *result = (float8 *) palloc(sizeof(float8));

    *result = *arg + 1.0;

    return result;
}

Point *
makepoint(Point *pointx, Point *pointy)
{
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    return new_point;
}

/* By Reference, Variable Length */

text *
copytext(text *t)
{
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text *new_t = (text *) palloc(VARSIZE(t));
    VARATT_SIZEP(new_t) = VARSIZE(t);
    /*
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t)-VARHDRSZ);    /* how many bytes */
    return new_t;
}

text *
concat_text(text *arg1, text *arg2)
{
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    VARATT_SIZEP(new_text) = new_text_size;
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1)-VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1)-VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2)-VARHDRSZ);
    return new_text;
}
```

Supposing that the above code has been prepared in file `funcs.c` and compiled into a shared object, we could define the functions to Postgres with commands like this:

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

-- note overloading of SQL function name add_one()
CREATE FUNCTION add_one(float8) RETURNS float8
     AS 'PGROOT/tutorial/funcs.so',
        'add_one_float8'
     LANGUAGE 'c' WITH (isStrict);

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

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

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

Here *`PGROOT`* stands for the full path to the Postgres source tree. Note that depending on your system, the filename for a shared object might not end in `.so`, but in `.sl` or something else; adapt accordingly.

Notice that we have specified the functions as "strict", meaning that the system should automatically assume a NULL result if any input value is NULL. By doing this, we avoid having to check for NULL inputs in the function code. Without this, we'd have to check for NULLs explicitly, for example by checking for a null pointer for each pass-by-reference argument. (For pass-by-value arguments, we don't even have a way to check!)

Although this calling convention is simple to use, it is not very portable; on some architectures there are problems with passing smaller-than-int data types this way. Also, there is no simple way to return a NULL result, nor to cope with NULL arguments in any way other than making the function strict. The version-1 convention, presented next, overcomes these objections.

13.4.3. Version-1 Calling Conventions for C-Language Functions

The version-1 calling convention relies on macros to suppress most of the complexity of passing arguments and results. The C declaration of a version-1 function is always

```
Datum funcname(PG_FUNCTION_ARGS)
```

In addition, the macro call

```
PG_FUNCTION_INFO_V1(funcname);
```

must appear in the same source file (conventionally it’s written just before the function itself). This macro call is not needed for "internal"-language functions, since Postgres currently assumes all internal functions are version-1. However, it is *required* for dynamically-loaded functions.

In a version-1 function, each actual argument is fetched using a `PG_GETARG_xxx()` macro that corresponds to the argument’s datatype, and the result is returned using a `PG_RETURN_xxx()` macro for the return type.

Here we show the same functions as above, coded in version-1 style:

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

/* By Value */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* By Reference, Fixed Length */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
    /* The macros for FLOAT8 hide its pass-by-reference nature */
    float8   arg = PG_GETARG_FLOAT8(0);
    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* Here, the pass-by-reference nature of Point is not hidden */
    Point     *pointx = PG_GETARG_POINT_P(0);
    Point     *pointy = PG_GETARG_POINT_P(1);
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* By Reference, Variable Length */

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_P(0);
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text     *new_t = (text *) palloc(VARSIZE(t));
    VARATT_SIZEP(new_t) = VARSIZE(t);
    /*
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t)-VARHDRSZ);    /* how many bytes */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text  *arg1 = PG_GETARG_TEXT_P(0);
    text  *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    VARATT_SIZEP(new_text) = new_text_size;
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1)-VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1)-VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2)-VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

The **CREATE FUNCTION** commands are the same as for the version-0 equivalents.

At first glance, the version-1 coding conventions may appear to be just pointless obscurantism. However, they do offer a number of improvements, because the macros can hide unnecessary detail. An example is that in coding add_one_float8, we no longer need to be aware that float8 is a pass-by-reference type. Another example is that the GETARG macros for variable-length types hide the need to deal with fetching "toasted" (compressed or out-of-line) values. The old-style `copytext` and `concat_text` functions shown above are actually wrong in the presence of toasted values, because they don't call `pg_detoast_datum()` on their inputs. (The handler for old-style dynamically-loaded functions currently takes care of this detail, but it does so less efficiently than is possible for a version-1 function.)

One big improvement in version-1 functions is better handling of NULL inputs and results. The macro `PG_ARGISNULL(n)` allows a function to test whether each input is NULL (of course, doing this is only necessary in functions not declared strict). As with the `PG_GETARG_xxx()` macros, the input arguments are counted beginning at zero. To return a NULL result, execute `PG_RETURN_NULL()`; this works in both strict and non-strict functions.

The version-1 function call conventions make it possible to return set results and implement trigger functions and procedural-language call handlers. Version-1 code is also more portable than version-0, because it does not break ANSI C restrictions on function call protocol. For more details see `src/backend/utils/fmgr/README` in the source distribution.

13.4.4. Composite Types in C-Language Functions

Composite types do not have a fixed layout like C structures. Instances of a composite type may contain null fields. In addition, composite types that are part of an inheritance hierarchy may have different fields than other members of the same inheritance hierarchy. Therefore, Postgres provides a procedural interface for accessing fields of composite types from C. As Postgres processes a set of rows, each row will be passed into your function as an opaque structure of type `TUPLE`. Suppose we want to write a function to answer the query

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
FROM emp
WHERE name = 'Bill' OR name = 'Sam';
```

In the query above, we can define c_overpaid as:

```
#include "postgres.h"
#include "executor/executor.h"  /* for GetAttributeByName() */

bool
c_overpaid(TupleTableSlot *t, /* the current row of EMP */
           int32 limit)
{
    bool isnull;
    int32 salary;

    salary = DatumGetInt32(GetAttributeByName(t, "salary", &isnull));
    if (isnull)
        return (false);
    return salary > limit;
}

/* In version-1 coding, the above would look like this: */

PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
    TupleTableSlot  *t = (TupleTableSlot *) PG_GETARG_POINTER(0);
    int32            limit = PG_GETARG_INT32(1);
    bool isnull;
    int32 salary;

    salary = DatumGetInt32(GetAttributeByName(t, "salary", &isnull));
    if (isnull)
        PG_RETURN_BOOL(false);
    /* Alternatively, we might prefer to do PG_RETURN_NULL() for null salary
*/

    PG_RETURN_BOOL(salary > limit);
}
```

`GetAttributeByName` is the Postgres system function that returns attributes out of the current row. It has three arguments: the argument of type `TupleTableSlot*` passed into the function, the name of the desired attribute, and a return parameter that tells whether the attribute is null. `GetAttributeByName` returns a Datum value that you can convert to the proper datatype by using the appropriate `DatumGet`*`XXX`*`()` macro.

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 rows or modify existing rows from within a C function, these are far too complex to discuss in this manual.

13.4.5. 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.

The basic rules for building C functions are as follows:

The relevant header (include) files are installed under `/usr/local/pgsql/include` or equivalent. You can use `pg_config --includedir` to find out where it is on your system (or the system that your users will be running on). For very low-level work you might need to have a complete PostgreSQL source tree available.

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`, while the function manager interfaces (PG_FUNCTION_ARGS, etc.) are in `fmgr.h`, so you will need to include at least these two files. For portability reasons it's best to include `postgres.h` *first*, before any other system or user header files. Including `postgres.h` will also include `c.h`, `elog.h` and `palloc.h` for you.

Symbol names defined within object files must not conflict with each other or with symbols defined in the PostgreSQL server executable. You will have to rename your functions or variables if you get error messages to this effect.

Compiling and linking your object code so that it can be dynamically loaded into Postgres always requires special flags. See Section 13.4.6 for a detailed explanation of how to do it for your particular operating system.

13.4.6. Compiling and Linking Dynamically-Loaded Functions

Before you are able to use your PostgreSQL extension function written in C they need to be compiled and linked in a special way in order to allow it to be dynamically loaded as needed by the server. To be precise, a *shared library* needs to be created.

For more information you should read the documentation of your operating system, in particular the manual pages for the C compiler, **cc**, and the link editor, **ld**. In addition, the PostgreSQL source code contains several working examples in the `contrib` directory. If you rely on these examples you will make your modules dependent on the availability of the PostgreSQL source code, however.

Creating shared libraries is generally analoguous to linking executables: first the source files are compiled into object files, then the object files are linked together. The object files need to be created as *position-independent code* (PIC), which conceptually means that they can be placed at an arbitrary location in memory when they are loaded by the executable. (Object files intended for executables are not compiled that way.) The command to link a shared library contains special flags to distinguish it from linking an executable. --- At least this is the theory. On some systems the practice is much uglier.

In the following examples we assume that your source code is in a file `foo.c` and we will create an shared library `foo.so`. The intermediate object file will be called `foo.o` unless otherwise noted. A shared library can contain more than one object file, but we only use one here.

BSD/OS

The compiler flag to create PIC is `-fpic`. The linker flag to create shared libraries is `-shared`.

```
gcc -fpic -c foo.c
ld -shared -o foo.so foo.o
```

This is applicable as of version 4.0 of BSD/OS.

FreeBSD

The compiler flag to create PIC is `-fpic`. To create shared libraries the compiler flag is `-shared`.

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

This is applicable as of version 3.0 of FreeBSD.

HP-UX

The compiler flag of the system compiler to create PIC is `+z`. When using GCC it's `-fpic`. The linker flag for shared libraries is `-b`. So

```
cc +z -c foo.c
```

or

```
gcc -fpic -c foo.c
```

and then

```
ld -b -o foo.sl foo.o
```

HP-UX uses the extension `.sl` for shared libraries, unlike most other systems.

Irix

PIC is the default, no special compiler options are necessary. The linker option to produce shared libraries is `-shared`.

```
cc -c foo.c
ld -shared -o foo.so foo.o
```

Linux

The compiler flag to create PIC is `-fpic`. On some platforms in some situations `-fPIC` must be used if `-fpic` does not work. Refer to the GCC manual for more information. The compiler flag to create a shared library is `-shared`. A complete example looks like this:

```
cc -fpic -c foo.c
cc -shared -o foo.so foo.o
```

NetBSD

The compiler flag to create PIC is `-fpic`. For ELF systems, the compiler with the flag `-shared` is used to link shared libraries. On the older non-ELF systems, `ld -Bshareable` is used.

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

The compiler flag to create PIC is `-fpic`. `ld -Bshareable` is used to link shared libraries.

```
gcc -fpic -c foo.c
ld -Bshareable -o foo.so foo.o
```

Digital Unix/Tru64 UNIX

PIC is the default, so the compilation command is the usual one. **ld** with special options is used to do the linking:

```
cc -c foo.c
ld -shared -expect_unresolved '*' -o foo.so foo.o
```

The same procedure is used with GCC instead of the system compiler; no special options are required.

Solaris

The compiler flag to create PIC is `-KPIC` with the Sun compiler and `-fpic` with GCC. To link shared libraries, the compiler option is `-G` with either compiler or alternatively `-shared` with GCC.

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

or

```
gcc -fpic -c foo.c
gcc -G -o foo.so foo.o
```

Unixware

The compiler flag to create PIC is `-K PIC` with the SCO compiler and `-fpic` with GCC. To link shared libraries, the compiler option is `-G` with the SCO compiler and `-shared` with GCC.

```
cc -K PIC -c foo.c
cc -G -o foo.so foo.o
```

or

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

Tip: If you want to package your extension modules for wide distribution you should consider using GNU Libtool (http://www.gnu.org/software/libtool/) for building shared libraries. It encapsulates the platform differences into a general and powerful interface. Serious packaging also requires considerations about library versioning, symbol resolution methods, and other issues.

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

Note: Actually, Postgres does not care what you name the file as long as it is a shared library file.

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. 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 user id the Postgres server runs as must be able to traverse the path given to the **CREATE FUNCTION** command and be able to read the shared library file. (Making the file or a higher-level directory not readable and/or not executable by the postgres user is a common mistake.)

13.5. Function Overloading

More than one function may be defined with the same name, so 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.

13.5.1. Name Space Conflicts

As of Postgres 7.0, 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.

13.5.1.1. Pre-7.0

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 14. Extending SQL: Types

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

14.1. User-Defined Types

14.1.1. 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(ERROR, "complex_in: error in parsing %s", str);
        return NULL;
    }
    result = (Complex *)palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    return (result);
}
```

The output function can simply be:

```
char *
complex_out(Complex *complex)
```

```
{
    char *result;
    if (complex == NULL)
        return(NULL);
    result = (char *) palloc(60);
    sprintf(result, "(%g,%g)", complex->x, complex->y);
    return(result);
}
```

You should try to make the input and output functions inverses of each other. If you do not, you will have severe problems when you need to dump your data into a file and then read it back in (say, into someone else's database on another computer). This is a particularly common problem when floating-point numbers are involved.

To define the complex type, we need to create the two user-defined functions complex_in and complex_out before creating the type:

```
CREATE FUNCTION complex_in(opaque)
    RETURNS complex
    AS 'PGROOT/tutorial/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.

14.1.2. Large Objects

If the values of your datatype might exceed a few hundred bytes in size (in internal form), you should be careful to mark them TOASTable. To do this, the internal representation must follow the standard layout for variable-length data: the first four bytes must be an int32 containing the total length in bytes of the datum (including itself). Then, all your functions that accept values of the type must be careful to call pg_detoast_datum() on the supplied values --- after checking that the value is not NULL, if your function is not strict. Finally, select the appropriate storage option when giving the CREATE TYPE command.

Chapter 15. 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.

15.1. 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.

15.1.1. COMMUTATOR

The COMMUTATOR clause, if provided, names an operator that is the commutator of the operator being defined. We say that operator A is the commutator of operator B if (x A y) equals (y B x) for all possible input values x,y. Notice that B is also the commutator of A. For example, operators '<' and '>' for a particular 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.)

15.1.2. NEGATOR

The NEGATOR clause, if provided, names an operator that is the negator of the operator being defined. We say that operator A is the negator of operator B if both return boolean results and (x A y) equals NOT (x B y) for all possible inputs x,y. Notice that B is also the negator of A. For example, ’<’ and ’>=’ are a negator pair for most 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 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 NOT (x = y) to be simplified into x <> y. This comes up more often than you might think, because NOTs can be inserted as a consequence of other rearrangements.

Pairs of negator operators can be defined using the same methods explained above for commutator pairs.

15.1.3. RESTRICT

The RESTRICT clause, if provided, names a restriction selectivity estimation function for the operator (note that this is a function name, not an operator name). RESTRICT clauses only make sense for binary operators that return boolean. The idea behind a restriction selectivity estimator is to guess what fraction of the rows in a table will satisfy a WHERE-clause condition of the form

```
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 scalarltsel 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/utils/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/utils/adt/geo_selfuncs.c: areasel, positionsel, and contsel. At this writing these are just stubs, but you may want to use them (or even better, improve them) anyway.

15.1.4. JOIN

The JOIN clause, if provided, names a join selectivity estimation function for the operator (note that this is a function name, not an operator name). JOIN clauses only make sense for binary operators that return boolean. The idea behind a join selectivity estimator is to guess what fraction of the rows in a pair of tables will satisfy a WHERE-clause condition of the form

```
table1.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:

```
eqjoinsel       for =
  neqjoinsel      for <>
  scalarltjoinsel for < or <=
  scalargtjoinsel for > or >=
  areajoinsel     for 2D area-based comparisons
  positionjoinsel for 2D position-based comparisons
  contjoinsel     for 2D containment-based comparisons
```

15.1.5. HASHES

The HASHES clause, if present, tells the system that it is OK to use the hash join method for a join based on this operator. HASHES only makes sense for binary operators that return boolean, and in practice the operator had better be equality for some data type.

The assumption underlying hash join is that the join operator can only return TRUE for pairs of left and right values that hash to the same hash code. If two values get put in different hash buckets, the join will never compare them at all, implicitly assuming that the result of the join operator must be FALSE. So it never makes sense to specify HASHES for operators that do not represent equality.

In fact, logical equality is not good enough either; the operator had better represent pure bitwise equality, because the hash function will be computed on the memory representation of the values regardless of what the bits mean. For example, equality of time intervals is not bitwise equality; the interval equality operator considers two time intervals equal if they have the same duration, whether or not their endpoints are identical. What this means is that a join using "=" between interval fields would yield different results if implemented as a hash join than if implemented another way, because a large fraction of the pairs that should match will hash to different values and will never be compared by the hash join. But if the optimizer chose to use a different kind of join, all the pairs that the equality operator says are equal will be found. We don't want that kind of inconsistency, so we don't mark interval equality as hashable.

There are also machine-dependent ways in which a hash join might fail to do the right thing. For example, if your 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().

15.1.6. SORT1 and SORT2

The SORT clauses, if present, tell the system that it is permissible to use the merge join method for a join based on the current operator. Both must be specified if either is. The current operator must be equality for some pair of data types, and the SORT1 and SORT2 clauses name the ordering operator ('<' operator) for the left and right-side data types respectively.

Merge join is based on the idea of sorting the left and righthand tables into order and then scanning them in parallel. So, both data types must be capable of being fully ordered, and the join operator must be one that can only succeed for pairs of values that fall at the "same place" in the sort order. In practice this means that the join operator must behave like equality. But unlike 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 16. 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. A *final function* can also be specified, in case the desired output of the aggregate is different from the data that needs to be kept in the running state value.

Thus, in addition to the input and result datatypes seen by a user of the aggregate, there is an internal state-value datatype that may be different from both the input and result types.

If we define an aggregate that does not use a final function, we have an aggregate that computes a running function of the column values from each row. "Sum" is an example of this kind of aggregate. "Sum" starts at zero and always adds the current row’s value to its running total. For example, if we want to make a Sum aggregate to work on a datatype for complex numbers, we only need the addition function for that datatype. The aggregate definition is:

```
CREATE AGGREGATE complex_sum (
    sfunc = complex_add,
    basetype = complex,
    stype = complex,
    initcond = ’(0,0)’
);

SELECT complex_sum(a) FROM test_complex;

         +------------+
         |complex_sum |
         +------------+
         |(34,53.9)   |
         +------------+
```

(In practice, we’d just name the aggregate "sum", and rely on Postgres to figure out which kind of sum to apply to a complex column.)

The above definition of "Sum" will return zero (the initial state condition) if there are no non-null input values. Perhaps we want to return NULL in that case instead --- SQL92 expects "Sum" to behave that way. We can do this simply by omitting the "initcond" phrase, so that the initial state condition is NULL. Ordinarily this would mean that the sfunc would need to check for a NULL state-condition input, but for "Sum" and some other simple aggregates like "Max" and "Min", it’s sufficient to insert the first non-null input value into the state variable and then start applying the transition function at the second non-null input value. Postgres will do that automatically if the initial condition is NULL and the transition function is marked "strict" (i.e., not to be called for NULL inputs).

Another bit of default behavior for a "strict" transition function is that the previous state value is retained unchanged whenever a NULL input value is encountered. Thus, NULLs are ignored. If you need some other behavior for NULL inputs, just define your transition function as non-strict, and code it to test for NULL inputs and do whatever is needed.

"Average" is a more complex example of an aggregate. It requires two pieces of running state: the sum of the inputs and the count of the number of inputs. The final result is obtained by dividing these quantities. Average is typically implemented by using a two-element array as the transition state value. For example, the built-in implementation of `avg(float8)` looks like:

```
CREATE AGGREGATE avg (
    sfunc = float8_accum,
    basetype = float8,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0}'
);
```

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

Chapter 17. The Postgres Rule System

Author: Written by Jan Wieck. Updates for 7.1 by Tom Lane.

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 planner for planning and execution. It is very powerful, and can be used for many things such as query language procedures, views, and versions. The power of this rule system is discussed in [*Ong and Goh, 1990*] as well as [*Stonebraker et al, ACM, 1990*].

17.1. What is a Querytree?

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

The rule system is located between the query parser and the planner. It takes the output of the parser, one 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 its input and output are always things the parser itself could have produced and thus, anything it sees is basically representable as an SQL statement.

Now what is a 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.

17.1.1. 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 these 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. It is expanded by the parser into the individual attributes, so the rule system never sees it.)

DELETE queries don't need a targetlist because they don't produce any result. In fact the planner will add a special CTID 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. It is the expressions in the VALUES clause or the ones from the SELECT clause in INSERT ... SELECT. Missing columns of the resultrelation will be filled in by the planner with a constant NULL expression.

In UPDATE queries, the targetlist describes the new rows that should replace the old ones. In the rule system, it contains just the expressions from the SET attribute = expression part of the query. The planner will add missing columns by inserting expressions that copy the values from the old row into the new one. And it will add the special CTID entry just as for DELETE too.

Every entry in the 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 query's 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 join tree

The query's join tree shows the structure of the FROM clause. For a simple query like SELECT FROM a, b, c the join tree is just a list of the FROM items, because we are allowed to join them in any order. But when JOIN expressions --- particularly outer joins --- are used, we have to join in the order shown by the JOINs. The join tree shows the structure of the JOIN expressions. The restrictions associated with particular JOIN clauses (from ON or USING expressions) are stored as qualification expressions attached to those join tree nodes. It turns out to be convenient to store the top-level WHERE expression as a qualification attached to the top-level join tree item, too. So really the join tree represents both the FROM and WHERE clauses of a SELECT.

the others

The other parts of the 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.

17.2. Views and the Rule System

17.2.1. 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 parser, there is absolutely no difference between a table and a view. They are the same thing - relations. That is the important one for now.

17.2.2. How SELECT Rules Work

Rules ON SELECT are applied to all queries as the last step, even if the command given is an INSERT, UPDATE or DELETE. And they have different semantics from the others in that they modify the parsetree in place instead of creating a new one. So SELECT rules are described first.

Currently, there can be only one action in an ON SELECT rule, and it must be an unconditional SELECT action that is INSTEAD. This restriction was required to make rules safe enough to open them for ordinary users and it restricts rules ON SELECT to real view rules.

The examples for this document are two join views that do some calculations and some more views using them in turn. One of the two first views is customized later by adding rules for INSERT, UPDATE and DELETE operations so that the final result will be a view that behaves like a real table with some magic functionality. It is not such a simple example to start from and this makes things harder to get into. But it’s better to have one example that covers all the points discussed step by step rather than having many different ones that might mix up in mind.

The database needed to play with the examples is named al_bundy. You’ll see soon why this is the database name. And it needs the procedural language PL/pgSQL installed, because we need a little min() function returning the lower of 2 integer values. We create that as

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS
    ’BEGIN
        IF $1 < $2 THEN
            RETURN $1;
        END IF;
        RETURN $2;
    END;’
LANGUAGE ’plpgsql’;
```

The real tables we need in the first two rule system descriptions are these:

```
CREATE TABLE shoe_data (
    shoename   char(10),      -- primary key
    sh_avail   integer,       -- available # of pairs
    slcolor    char(10),      -- preferred shoelace color
    slminlen   float,         -- miminum shoelace length
    slmaxlen   float,         -- maximum shoelace length
    slunit     char(8)        -- length unit
);

CREATE TABLE shoelace_data (
    sl_name    char(10),      -- primary key
    sl_avail   integer,       -- available # of pairs
    sl_color   char(10),      -- shoelace color
    sl_len     float,         -- shoelace length
    sl_unit    char(8)        -- length unit
);

CREATE TABLE unit (
    un_name    char(8),       -- the primary key
    un_fact    float          -- factor to transform to cm
);
```

I think most of us wear shoes and can realize that this is really useful data. Well there are shoes out in the world that don’t require shoelaces, but this doesn’t make Al’s life easier and so we ignore it.

The views are created as

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           min(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
     WHERE rsl.sl_color = rsh.slcolor
       AND rsl.sl_len_cm >= rsh.slminlen_cm
       AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

The CREATE VIEW command for the `shoelace` view (which is the simplest one we have) will create a relation shoelace and an entry in `pg_rewrite` that tells that there is a rewrite rule that must be applied whenever the relation shoelace is referenced in a query's rangetable. The rule has no rule qualification (discussed later, with the non SELECT rules, since SELECT rules currently cannot have them) and it is INSTEAD. Note that rule qualifications are not the same as query qualifications! The rule's action has a query qualification.

The rule's action is one querytree that is a copy of the SELECT statement in the view creation command.

> **Note:** The two extra range table entries for NEW and OLD (named *NEW* and *CURRENT* for historical reasons in the printed 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->     ('sl4', 8, 'black', 40.0 , 'inch');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl5', 4, 'brown', 1.0 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl6', 0, 'brown', 0.9 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl7', 7, 'brown', 60 , 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl8', 1, 'brown', 40 , 'inch');
al_bundy=>
al_bundy=> SELECT * FROM shoelace;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       7|brown     |    60|cm      |       60
sl3       |       0|black     |    35|inch    |     88.9
sl4       |       8|black     |    40|inch    |    101.6
sl8       |       1|brown     |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |       0|brown     |   0.9|m       |       90
(8 rows)
```

It's the simplest SELECT Al can do on our views, so we take this to explain the basics of view rules. The 'SELECT * FROM shoelace' was interpreted by the parser and produced the 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.

To expand the view, the rewriter simply creates a subselect rangetable entry containing the rule's action parsetree, and substitutes this rangetable entry for the original one that referenced the view. The resulting rewritten parsetree is almost the same as if Al had typed

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM (SELECT s.sl_name,
               s.sl_avail,
               s.sl_color,
               s.sl_len,
               s.sl_unit,
               s.sl_len * u.un_fact AS sl_len_cm
          FROM shoelace_data s, unit u
         WHERE s.sl_unit = u.un_name) shoelace;
```

There is one difference however: the sub-query's rangetable has two extra entries shoelace *OLD*, shoelace *NEW*. These entries don't participate directly in the query, since they aren't referenced by the sub-query's join tree or targetlist. The rewriter uses them to store the access permission check info that was originally present in the rangetable entry that referenced the view. In this way, the executor will still check that the user has proper permissions to access the view, even though there's no direct use of the view in the rewritten query.

That was the first rule applied. The rule system will continue checking the remaining rangetable entries in the top query (in this example there are no more), and it will recursively check the rangetable entries in the added sub-query to see if any of them reference views. (But it won't expand *OLD* or *NEW* --- otherwise we'd have infinite recursion!) In this example, there are no rewrite rules for shoelace_data or unit, so rewriting is complete and the above is the final result given to the planner.

Now we face Al with the problem that the Blues Brothers appear in his shop and want to buy some new shoes, and as the Blues Brothers are, they want to wear the same shoes. And they want to wear them immediately, so they need shoelaces too.

Al needs to know for which shoes currently in the store he has the matching shoelaces (color and size) and where the total number of exactly matching pairs is greater or equal to two. We teach him what to do and he asks his database:

```
al_bundy=> SELECT * FROM shoe_ready WHERE total_avail >= 2;
shoename  |sh_avail|sl_name   |sl_avail|total_avail
----------+--------+----------+--------+-----------
sh1       |       2|sl1       |       5|          2
sh3       |       4|sl7       |       7|          4
(2 rows)
```

Al is a shoe guru and so he knows that only shoes of type sh1 would fit (shoelace sl7 is brown and shoes that need brown shoelaces aren't shoes the Blues Brothers would ever wear).

The output of the parser this time is the 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 the one for the `shoe_ready` view and it results in the parsetree

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM shoe rsh, shoelace rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE int4ge(shoe_ready.total_avail, 2);
```

Similarly, the rules for `shoe` and `shoelace` are substituted into the rangetable of the sub-query, leading to a three-level final querytree:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM (SELECT sh.shoename,
                       sh.sh_avail,
                       sh.slcolor,
                       sh.slminlen,
                       sh.slminlen * un.un_fact AS slminlen_cm,
                       sh.slmaxlen,
                       sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                       sh.slunit
                  FROM shoe_data sh, unit un
                 WHERE sh.slunit = un.un_name) rsh,
               (SELECT s.sl_name,
                       s.sl_avail,
                       s.sl_color,
                       s.sl_len,
                       s.sl_unit,
                       s.sl_len * u.un_fact AS sl_len_cm
                  FROM shoelace_data s, unit u
                 WHERE s.sl_unit = u.un_name) rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE int4ge(shoe_ready.total_avail, 2);
```

It turns out that the planner will collapse this tree into a two-level querytree: the bottommost selects will be "pulled up" into the middle select since there's no need to process them separately. But the middle select will remain separate from the top, because it contains aggregate functions. If we pulled those up it would change the behavior of the topmost select, which we don't want. However, collapsing the query tree is an optimization that the rewrite system doesn't have to concern itself with.

> **Note:** There is currently no recursion stopping mechanism for view rules in the rule system (only for the other kinds of rules). This doesn't hurt much, because the only way to push this into an endless loop (blowing up the backend until it reaches the memory limit) is to create tables and then setup the view rules by hand with CREATE RULE in such a way, that one selects from the other that selects from the one. This could never happen if CREATE VIEW is used because for the first CREATE VIEW, the second relation does not exist and thus the first view cannot select from the second.

17.2.3. 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 this information.

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. Everything 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 jointrees show a simple join between t1 and t2.

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 planner 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 rows that are there has to be replaced by the new row?

To resolve this problem, another entry is added to the targetlist in UPDATE (and also in DELETE) statements: the current tuple ID (ctid). This is a system attribute containing the file block number and position in the block for the row. Knowing the table, the ctid can be used to retrieve the original t1 row to be updated. After adding the ctid to the targetlist, the query actually looks like

```
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 commited the vacuum cleaner can really move it out.

Knowing all that, we can simply apply view rules in absolutely the same way to any command. There is no difference.

17.2.4. 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).

17.2.4.1. Benefits

The benefit of implementing views with the rule system is, that the planner has all the information about which tables have to be scanned plus the relationships between these tables plus the restrictive qualifications from the views plus the qualifications from the original query in one single parsetree. And this is still the situation when the original query is already a join over views. Now the planner has to decide which is the best path to execute the query. The more information the planner has, the better this decision can be. And the rule system as implemented in Postgres ensures, that this is all information available about the query up to now.

17.2.5. What about updating a view?

What happens if a view is named as the target relation for an INSERT, UPDATE, or DELETE? After doing the substitutions described above, we will have a querytree in which the resultrelation points at a subquery rangetable entry. This will not work, so the rewriter throws an error if it sees it has produced such a thing.

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

17.3. Rules on INSERT, UPDATE and DELETE

17.3.1. Differences from View Rules

Rules that are defined ON INSERT, UPDATE and DELETE are totally different from the view rules described in the previous section. First, their CREATE RULE command allows more:

They can have no action.

They can have multiple actions.

The keyword INSTEAD is optional.

The pseudo relations NEW and OLD become useful.

They can have rule qualifications.

Second, they don't modify the parsetree in place. Instead they create zero or many new parsetrees and can throw away the original one.

17.3.2. How These Rules Work

Keep the syntax

```
CREATE RULE rule_name AS ON event
    TO object [WHERE rule_qualification]
    DO [INSTEAD] [action | (actions) | NOTHING];
```

in mind. In the following, "update rules" means rules that are defined ON INSERT, UPDATE or DELETE.

Update rules get applied by the rule system when the result relation and the 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 parsetree's qualification has been added.

No qualification but INSTEAD:

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

Qualification given and not INSTEAD:

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

Qualification given and INSTEAD:

The parsetree from the rule action where the rule qualification and the original parsetree's 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 either one or two output 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 entry's expression replaces the reference. Otherwise NEW means the same as OLD (for an UPDATE) or is replaced by NULL (for an INSERT). Any reference to OLD is replaced by a reference to the rangetable entry which is the resultrelation.

After we are done applying update rules, we apply view rules to the produced parsetree(s). Views cannot insert new update actions so there is no need to apply update rules to the output of view rewriting.

17.3.2.1. A First Rule Step by Step

We want to trace changes to the sl_avail column in the `shoelace_data` relation. So we setup a log table and a rule that conditionally writes a log entry when an UPDATE is performed on `shoelace_data`.

```
CREATE TABLE shoelace_log (
    sl_name    char(10),      -- shoelace changed
    sl_avail   integer,       -- new available value
    log_who    text,          -- who did it
    log_when   timestamp      -- when
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail != OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

Now Al does

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

and we look at the 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 VALUES(
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp
  FROM shoelace_data *NEW*, shoelace_data *OLD*;
```

This is a little strange-looking since you can't normally write INSERT ... VALUES ... FROM. The FROM clause here is just to indicate that there are rangetable entries in the parsetree for *NEW* and *OLD*. These are needed so that they can be referenced by variables in the INSERT command's querytree.

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

```
INSERT INTO shoelace_log VALUES(
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data;
```

In step 2 the rule qualification is added to it, so the result set is restricted to rows where sl_avail changes.

```
    INSERT INTO shoelace_log VALUES(
           *NEW*.sl_name, *NEW*.sl_avail,
           current_user, current_timestamp
      FROM shoelace_data *NEW*, shoelace_data *OLD*,
           shoelace_data shoelace_data
     WHERE int4ne(*NEW*.sl_avail, *OLD*.sl_avail);
```

This is even stranger-looking, since INSERT ... VALUES doesn't have a WHERE clause either, but the planner and executor will have no difficulty with it. They need to support this same functionality anyway for INSERT ... SELECT. In step 3 the original parsetree's qualification is added, restricting the resultset further to only the rows touched by the original parsetree.

```
    INSERT INTO shoelace_log VALUES(
           *NEW*.sl_name, *NEW*.sl_avail,
           current_user, current_timestamp
      FROM shoelace_data *NEW*, shoelace_data *OLD*,
           shoelace_data shoelace_data
     WHERE int4ne(*NEW*.sl_avail, *OLD*.sl_avail)
       AND bpchareq(shoelace_data.sl_name, 'sl7');
```

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

```
    INSERT INTO shoelace_log VALUES(
           shoelace_data.sl_name, 6,
           current_user, current_timestamp
      FROM shoelace_data *NEW*, shoelace_data *OLD*,
           shoelace_data shoelace_data
     WHERE int4ne(6, *OLD*.sl_avail)
       AND bpchareq(shoelace_data.sl_name, 'sl7');
```

Step 5 changes OLD references into resultrelation references.

```
    INSERT INTO shoelace_log VALUES(
           shoelace_data.sl_name, 6,
           current_user, current_timestamp
      FROM shoelace_data *NEW*, shoelace_data *OLD*,
           shoelace_data shoelace_data
     WHERE int4ne(6, shoelace_data.sl_avail)
       AND bpchareq(shoelace_data.sl_name, 'sl7');
```

That's it. Since the rule is not INSTEAD, we also output the original parsetree. In short, the output from the rule system is a list of two parsetrees that are the same as the statements:

```
    INSERT INTO shoelace_log VALUES(
           shoelace_data.sl_name, 6,
           current_user, current_timestamp
      FROM shoelace_data
     WHERE 6 != shoelace_data.sl_avail
```

```
       AND shoelace_data.sl_name = 'sl7';

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

These are executed in this order and that is exactly what the rule defines. The substitutions and the qualifications added ensure that if the original query would be, say,

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

no log entry would get written. This time the original parsetree does not contain a targetlist entry for sl_avail, so NEW.sl_avail will get replaced by shoelace_data.sl_avail resulting in the extra query

```
    INSERT INTO shoelace_log VALUES(
           shoelace_data.sl_name, shoelace_data.sl_avail,
           current_user, current_timestamp)
      FROM shoelace_data
     WHERE shoelace_data.sl_avail != shoelace_data.sl_avail
       AND shoelace_data.sl_name = 'sl7';
```

and that qualification will never be true. It will also work if the original query modifies multiple rows. So if Al would issue the command

```
    UPDATE shoelace_data SET sl_avail = 0
     WHERE sl_color = 'black';
```

four rows in fact get updated (sl1, sl2, sl3 and sl4). But sl3 already has sl_avail = 0. This time, the original parsetrees qualification is different and that results in the extra parsetree

```
    INSERT INTO shoelace_log SELECT
           shoelace_data.sl_name, 0,
           current_user, current_timestamp
      FROM shoelace_data
     WHERE 0 != shoelace_data.sl_avail
       AND shoelace_data.sl_color = 'black';
```

This 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.

17.3.3. Cooperation with Views

A simple way to protect view relations from the mentioned possibility that someone can try to INSERT, UPDATE and DELETE on them is to let those 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 Al now tries to do any of these operations on the view relation `shoe`, the rule system will apply the rules. Since the rules have no actions and are INSTEAD, the resulting list of 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 way might irritate frontend applications because absolutely nothing happened on the database and thus, the backend will not return anything for the query. Not even a PGRES_EMPTY_QUERY will be available in libpq. In psql, nothing happens. This might change in the future.

A more sophisticated way to use the rule system is to create rules that rewrite the 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 these are:

```
CREATE TABLE shoelace_arrive (
    arr_name    char(10),
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     char(10),
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace SET
           sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

Now Al can sit down and do whatever until

```
al_bundy=> SELECT * FROM shoelace_arrive;
arr_name  |arr_quant
----------+---------
sl3       |       10
sl6       |       20
sl8       |       20
(3 rows)
```

is exactly what's on the part list. We take a quick look at the current data,

```
al_bundy=> SELECT * FROM shoelace 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 its 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 still uses the view `shoelace`. But the rule system isn't finished with this loop so it continues and applies the rule '_RETshoelace' on it and we get

```
UPDATE shoelace_data SET
       sl_name = s.sl_name,
       sl_avail = int4pl(s.sl_avail, shoelace_arrive.arr_quant),
       sl_color = s.sl_color,
       sl_len = s.sl_len,
       sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data showlace_data,
       shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u
 WHERE bpchareq(s.sl_name, showlace_arrive.arr_name)
   AND bpchareq(shoelace_data.sl_name, s.sl_name);
```

Again an update rule has been applied and so the wheel turns on and we are in rewrite round 3. This time rule 'log_shoelace' gets applied what produces the extra parsetree

```
    INSERT INTO shoelace_log SELECT
           s.sl_name,
           int4pl(s.sl_avail, shoelace_arrive.arr_quant),
           current_user,
           current_timestamp
      FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
           shoelace_ok *OLD*, shoelace_ok *NEW*,
           shoelace shoelace, shoelace *OLD*,
           shoelace *NEW*, shoelace_data showlace_data,
           shoelace *OLD*, shoelace *NEW*,
           shoelace_data s, unit u,
           shoelace_data *OLD*, shoelace_data *NEW*
           shoelace_log shoelace_log
     WHERE bpchareq(s.sl_name,  showlace_arrive.arr_name)
       AND bpchareq(shoelace_data.sl_name, s.sl_name);
       AND int4ne(int4pl(s.sl_avail, shoelace_arrive.arr_quant),
                                                  s.sl_avail);
```

After that the rule system runs out of rules and returns the generated 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,
           current_user,
           current_timestamp
      FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
           shoelace_data s
     WHERE s.sl_name = shoelace_arrive.arr_name
       AND shoelace_data.sl_name = s.sl_name
       AND s.sl_avail + shoelace_arrive.arr_quant != s.sl_avail;

    UPDATE shoelace_data SET
           sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
     FROM shoelace_arrive shoelace_arrive,
          shoelace_data shoelace_data,
          shoelace_data s
    WHERE s.sl_name = shoelace_arrive.sl_name
      AND shoelace_data.sl_name = s.sl_name;
```

The result is that data coming from one relation inserted into another, changed into updates on a third, changed into updating a fourth plus logging that final update in a fifth gets reduced into two queries.

There is a little detail that's a bit ugly. Looking at the two queries turns out, that the `shoelace_data` relation appears twice in the rangetable where it could definitely be reduced to one. The planner does not handle it and so the execution plan for the rule systems output of the INSERT will be

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

while omitting the extra 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->     ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
    al_bundy=> INSERT INTO shoelace VALUES
    al_bundy->     ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

Since this happens often, we must lookup for shoelace entries, that fit for absolutely no shoe sometimes. We could do that in a complicated statement every time, or we can setup a view for it. The view for this is

```
    CREATE VIEW shoelace_obsolete AS
        SELECT * FROM shoelace WHERE NOT EXISTS
            (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

Its output is

```
    al_bundy=> SELECT * FROM shoelace_obsolete;
    sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
    ----------+--------+----------+------+--------+---------
```

```
sl9       |       0|pink      |    35|inch    |     88.9
sl10      |    1000|magenta   |    40|inch    |    101.6
```

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

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

and do it this way:

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

Voila:

```
al_bundy=> SELECT * FROM shoelace;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       6|brown     |    60|cm      |       60
sl4       |       8|black     |    40|inch    |    101.6
sl3       |      10|black     |    35|inch    |     88.9
sl8       |      21|brown     |    40|inch    |    101.6
sl10      |    1000|magenta   |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |      20|brown     |   0.9|m       |       90
(9 rows)
```

A DELETE on a view, with a subselect qualification that in total uses 4 nesting/joined views, where one of them itself has a subselect qualification containing a view and where calculated view columns are used, gets rewritten into one single 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.

17.4. 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 against the permissions of the rule owner, not the user invoking the rule. This means, that a user does only need the required permissions for the tables/views he names in his queries.

For example: A user has a list of phone numbers where some of them are private, the others are of interest for the secretary of the office. He can construct the following:

```
CREATE TABLE phone_data (person text, phone text, private bool);
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE NOT private;
GRANT SELECT ON phone_number TO secretary;
```

Nobody except him (and the database superusers) can access the phone_data table. But due to the GRANT, the secretary can SELECT from the phone_number view. The rule system will rewrite the SELECT from phone_number into a SELECT from phone_data and add the qualification that only entries where private is false are wanted. Since the user is the owner of phone_number, the read access to phone_data is now checked against his permissions and the query is considered granted. The check for accessing phone_number is also performed, but this is done against the invoking user, so nobody but the user and the secretary can use it.

The permissions are checked rule by rule. So the secretary is for now the only one who can see the public phone numbers. But the secretary can setup another view and grant access to that to public. Then, anyone can see the phone_number data through the secretaries view. What the secretary cannot do is to create a view that directly accesses phone_data (actually he can, but it will not work since every access aborts the transaction during the permission checks). And as soon as the user will notice, that the secretary opened his phone_number view, he can REVOKE his access. Immediately any access to the secretaries view will fail.

Someone might think that this rule by rule checking is a security hole, but in fact it isn't. If this would not work, the secretary could setup a table with the same columns as phone_number and copy the data to there once per day. Then it's his own data and he can grant access to everyone he wants. A GRANT means "I trust you". If someone you trust does the thing above, it's time to think it over and then REVOKE.

This mechanism does also work for update rules. In the examples of the previous section, the owner of the tables in Al's database could GRANT SELECT, INSERT, UPDATE and DELETE on the shoelace view to al. But only SELECT on shoelace_log. The rule action to write log entries will still be executed successfully. And Al could see the log entries. But he cannot create fake entries, nor could he manipulate or remove existing ones.

Warning: GRANT ALL currently includes RULE permission. This means the granted user could drop the rule, do the changes and reinstall it. I think this should get changed quickly.

17.5. 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 planner will create a plan of

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

So there would be not that much difference in speed between the trigger and the rule implementation. With the next delete we want to get rid of all the 2000 computers where the hostname starts with ’old’. There are two possible queries to do that. One is

```
DELETE FROM computer WHERE hostname >= ’old’
```

```
                      AND hostname <  'ole'
```

Where the plan for the rule query will be a

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
        ->  Index Scan using comp_hostidx on computer
```

The other possible query is a

```
DELETE FROM computer WHERE hostname ~ '^old';
```

with the execution plan

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

This shows, that the planner does not realize that the qualification for the hostname on computer could also be used for an index scan on software when there are multiple qualification expressions combined with AND, what he does in the regexp version of the query. The trigger will get invoked once for any of the 2000 old computers that have to be deleted and that will result in one index scan over computer and 2000 index scans for the software. The rule implementation will do it with two queries over 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 manufacurer = 'bim';
```

Again this could result in many rows to be deleted from computer. So the trigger will again fire many queries into the executor. But the rule plan will again be the 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.manufacurer = 'bim'
                       AND software.hostname = computer.hostname;
```

In any of these cases, the extra queries from the rule system will be more or less independent from the number of affected rows in a query.

Another situation is cases on UPDATE where it depends on the change of an attribute if an action should be performed or not. In Postgres version 6.4, the attribute specification for rule events is disabled (it will have its comeback latest in 6.5, maybe earlier - stay tuned). So for now the only way to create a rule as in the shoelace_log example is to do it with a rule qualification. That results in an extra query that is performed always, even if the attribute of interest cannot change at all because it does not appear in the 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 its actions will only be done when a specific attribute is updated is hidden in its functionality. The definition of a trigger only allows to specify it on row level, so whenever a row is touched, the trigger must be called to make its decision. The rule system will know it by looking up the 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 significantly slower than triggers if their actions result in large and bad qualified joins, a situation where the planner fails. They are a big hammer. Using a big hammer without caution can cause big damage. But used with the right touch, they can hit any nail on the head.

Chapter 18. 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 Figure 12-1. 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` table contains one row 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 18-1. Index Schema

Column	Description
amname	name of the access method
amowner	user id of the owner
amstrategies	number of strategies for this access method (see below)
amsupport	number of support routines for this access method (see below)
amorderstrategy	zero if the index offers no sort order, otherwise the strategy number of the strategy operator that describes the sort order
amgettuple	
aminsert	
...	procedure identifiers for interface routines to the access method. For example, regproc ids for opening, closing, and getting rows from the access method appear here.

The object ID of the row in `pg_am` is used as a foreign key in a lot of other tables. You do not need to add a new rows to this table; all that you are interested in is the object ID of the access method row 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` column 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 (e.g., ">" or "<") and tell what kind of comparison it is. In fact, some access methods don't impose any ordering at all. For example, R-trees express a rectangle-containment relationship, whereas a hashed data structure expresses only bitwise similarity based on the value of a hash function. 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` table, the amstrategies column is the number of strategies defined for this access method. For B-trees, this number is 5. These strategies correspond to

Table 18-2. 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 a column called `amsupport`. This column records the number of support routines used by an access method. For B-trees, this number is one -- the routine to take two keys and return -1, 0, or +1, depending on whether the first key is less than, equal to, or greater than the second.

> **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 table of interest is `pg_opclass`. This table 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 a row with your opclass name (for example, `complex_abs_ops`) to `pg_opclass`. The `oid` of this row will be a foreign key in other tables, notably `pg_amop`.

```
INSERT INTO pg_opclass (opcname, opcdeftype)
    SELECT 'complex_abs_ops', oid FROM pg_type WHERE typname = '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` row will be different! Don't worry about this though. We'll get this number from the system later just like we got the oid of the type here.

The above example assumes that you want to make this new 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 column of the `pg_am` table. We will use this later on. For now, ignore it.

Now we are ready to define the operators:

```
CREATE OPERATOR = (
     leftarg = complex, rightarg = complex,
     procedure = complex_abs_eq,
     restrict = eqsel, join = eqjoinsel
         )
```

The important things here are the procedure names (which are the C functions defined above) and the restriction and join selectivity functions. You should just use the selectivity functions used in the example (see `complex.source`). Note that there are different such functions for the less-than, equal, and greater-than cases. These must be supplied, or the optimizer will be unable to make effective use of the index.

The next step is to add entries for these operators to the `pg_amop` relation. To do this, we'll need the `oid`s of the operators we just defined. We'll look up the names of all the operators that take two `complex`es, 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 `oid`s 277970 through 277974. The values you get will probably be different, and you should substitute them for the values below. We will do this with a select statement.

Now we are ready to 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 rows 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` table, 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 row 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 19. Index Cost Estimation Functions

Author: Written by Tom Lane (<tgl@sss.pgh.pa.us>) on 2000-01-24

Note: This must eventually become part of a much larger chapter about writing new index access methods.

Every index access method must provide a cost estimation function for use by the planner/optimizer. The procedure OID of this function is given in the `amcostestimate` field of the access method's `pg_am` entry.

Note: Prior to 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 start-up 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 "start-up cost" is the part of the total scan cost that must be expended before we can begin to fetch the first tuple. For most indexes this can be taken as zero, but an index type with a high start-up cost might want to set it nonzero.

The indexSelectivity should be set to the estimated fraction of the main table tuples that will be retrieved during the index scan. In the case of a lossy index, this will typically be higher than the fraction of tuples that actually pass the given qual conditions.

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

```
prorettype = 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 20. 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 http://epoch.cs.berkeley.edu:8000/ and http://www.sai.msu.su/~megera/postgres/gist/.

> **Author:** This extraction from an email sent by Eugene Selkov, Jr. (<selkovjr@mcs.anl.gov>) contains good information on GiST. Hopefully we will learn more in the future and update this information. - thomas 1998-03-01

Well, I can't say I quite understand what's going on, but at least I (almost) succeeded in porting GiST examples to linux. The GiST access method is already in the postgres tree (`src/backend/access/gist`).

Examples at Berkeley (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 that the original makefile.

2. I built the example sources right under pgsql/src (just extracted the tar file there). The aforementioned Makefile assumes it is one level below pgsql/src (in our case, in pgsql/src/pggist).

3. The changes I made to the *.c files were all about #include's, function prototypes and typecasting. Other than that, I just threw away a bunch of unused vars and added a couple parentheses to please gcc. I hope I did not screw up too much :)

4. There is a comment in polyproc.sql:

```
-- -- there’s a memory leak in rtree poly_ops!!
-- -- create index pix2 on polytmp using rtree (p poly_ops);
```

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 21. Triggers

Postgres has various server-side function interfaces. Server-side functions can be written in SQL, PLPGSQL, TCL, or C. Trigger functions can be written in any of these languages except SQL. Note that STATEMENT-level trigger events are not supported in the current version. You can currently specify BEFORE or AFTER on INSERT, DELETE or UPDATE of a tuple as a trigger event.

21.1. Trigger Creation

If a trigger event occurs, the trigger manager (called by the Executor) sets up a TriggerData information structure (described below) and calls the trigger function to handle the event.

The trigger function must be created before the trigger is created as a function taking no arguments and returning opaque. If the function is written in C, it must use the "version 1" function manager interface.

The syntax for creating triggers is as follows:

```
CREATE TRIGGER trigger [ BEFORE | AFTER ] [ INSERT | DELETE | UPDATE [ OR ...
] ]
    ON relation FOR EACH [ ROW | STATEMENT ]
    EXECUTE PROCEDURE procedure
     (args);
```

where the arguments are:

trigger

The name of the trigger is used if you ever have to delete the trigger. It is used as an argument to the **DROP TRIGGER** command.

BEFORE
AFTER

Determines whether the function is called before or after the event.

INSERT
DELETE
UPDATE

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

relation

The relation name determines which table the event applies to.

ROW
STATEMENT

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

procedure

The procedure name is the function called.

args

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

Also, *procedure* may be used for triggering different relations (these functions are named as "general trigger functions").

As example of using both features above, there could be a general function that takes as its arguments two field names and puts the current user in one and the current timestamp in the other. This allows triggers to be written on INSERT events to automatically track creation of records in a transaction table for example. It could also be used as a "last updated" function if used in an UPDATE event.

Trigger functions return HeapTuple to the calling Executor. This is ignored for triggers fired after an INSERT, DELETE or UPDATE operation but it allows BEFORE triggers to:

Return NULL to skip the operation for the current tuple (and so the tuple will not be inserted/updated/deleted).

Return a pointer to another tuple (INSERT and UPDATE only) which will be inserted (as the new version of the updated tuple if UPDATE) instead of original tuple.

Note that there is no initialization performed by the CREATE TRIGGER handler. This will be changed in the future. Also, if more than one trigger is defined for the same event on the same relation, the order of trigger firing is unpredictable. This may be changed in the future.

If a trigger function executes SQL-queries (using SPI) then these queries may fire triggers again. This is known as cascading triggers. There is no explicit limitation on the number of cascade levels.

If a trigger is fired by INSERT and inserts a new tuple in the same relation then this trigger will be fired again. Currently, there is nothing provided for synchronization (etc) of these cases but this may change. At the moment, there is function funny_dup17() in the regress tests which uses some techniques to stop recursion (cascading) on itself...

21.2. Interaction with the Trigger Manager

This section describes the low-level details of the interface to a trigger function. This information is only needed when writing a trigger function in C. If you are using a higher-level function language then these details are handled for you.

> **Note:** The interface described here applies for Postgres 7.1 and later. Earlier versions passed the TriggerData pointer in a global variable CurrentTriggerData.

When a function is called by the trigger manager, it is not passed any normal parameters, but it is passed a "context" pointer pointing to a TriggerData structure. C functions can check whether they were called from the trigger manager or not by executing the macro `CALLED_AS_TRIGGER(fcinfo)`, which expands to

```
        ((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

If this returns TRUE, then it is safe to cast fcinfo->context to type `TriggerData *` and make use of the pointed-to TriggerData structure. The function must *not* alter the TriggerData structure or any of the data it points to.

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

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

where the members are defined as follows:

type

Always `T_TriggerData` if this is a trigger event.

tg_event

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

TRIGGER_FIRED_BEFORE(tg_event)

returns TRUE if trigger fired BEFORE.

TRIGGER_FIRED_AFTER(tg_event)

Returns TRUE if trigger fired AFTER.

TRIGGER_FIRED_FOR_ROW(event)

Returns TRUE if trigger fired for a ROW-level event.

TRIGGER_FIRED_FOR_STATEMENT(event)

Returns TRUE if trigger fired for STATEMENT-level event.

TRIGGER_FIRED_BY_INSERT(event)

Returns TRUE if trigger fired by INSERT.

TRIGGER_FIRED_BY_DELETE(event)

Returns TRUE if trigger fired by DELETE.

TRIGGER_FIRED_BY_UPDATE(event)

Returns TRUE if trigger fired by UPDATE.

tg_relation

is a pointer to structure describing the triggered relation. Look at src/include/utils/rel.h for details about this structure. The most interest things are tg_relation->rd_att (descriptor of the relation tuples) and tg_relation->rd_rel->relname (relation’s name. This is not char*, but NameData. Use SPI_getrelname(tg_relation) to get char* if you need a copy of name).

tg_trigtuple

is a pointer to the tuple for which the trigger is fired. This is the tuple being inserted (if INSERT), deleted (if DELETE) or updated (if UPDATE). If INSERT/DELETE then this is what you are to return to Executor if you don’t want to replace tuple with another one (INSERT) or skip the operation.

tg_newtuple

is a pointer to the new version of tuple if UPDATE and NULL if this is for an INSERT or a DELETE. This is what you are to return to Executor if UPDATE and you don’t want to replace this tuple with another one or skip the operation.

tg_trigger

is pointer to structure Trigger defined in src/include/utils/rel.h:

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    FmgrInfo    tgfunc;
    int16       tgtype;
    bool        tgenabled;
    bool        tgisconstraint;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgattr[FUNC_MAX_ARGS];
    char      **tgargs;
} Trigger;
```

where tgname is the trigger's name, tgnargs is number of arguments in tgargs, tgargs is an array of pointers to the arguments specified in the CREATE TRIGGER statement. Other members are for internal use only.

21.3. 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 that are started after query Q, no matter whether they are started inside Q (during the execution of Q) or after Q is done.

This is true for triggers as well so, though a tuple being inserted (tg_trigtuple) is not visible to queries in a BEFORE trigger, this tuple (just inserted) is visible to queries in an AFTER trigger, and to queries in BEFORE/AFTER triggers fired after this!

21.4. Examples

There are more complex examples in `src/test/regress/regress.c` and in `contrib/spi`.

Here is a very simple example of trigger usage. Function trigf reports the number of tuples in the triggered relation ttest and skips the operation if the query attempts to insert NULL into x (i.e - it acts as a NOT NULL constraint but doesn't abort the transaction).

```
#include "executor/spi.h"   /* this is what you need to work with SPI */
#include "commands/trigger.h"   /* -"- and triggers */

extern Datum trigf(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
        TriggerData *trigdata = (TriggerData *) fcinfo->context;
        TupleDesc       tupdesc;
        HeapTuple       rettuple;
        char            *when;
        bool            checknull = false;
        bool            isnull;
        int             ret, i;
        /* Make sure trigdata is pointing at what I expect */
        if (!CALLED_AS_TRIGGER(fcinfo))
                elog(ERROR, "trigf: not fired by trigger manager");

        /* tuple to return to Executor */
        if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
                rettuple = trigdata->tg_newtuple;
        else
                rettuple = trigdata->tg_trigtuple;

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

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

        tupdesc = trigdata->tg_relation->rd_att;

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

        /* Get number of tuples in relation */
        ret = SPI_exec("select count(*) from ttest", 0);

        if (ret < 0)
                elog(NOTICE, "trigf (fired %s): SPI_exec returned %d", when,
ret);

         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 PointerGetDatum(rettuple);
}
```

Now, compile and create the trigger function:

```
create function trigf () returns opaque as
'...path_to_so' language 'C';

create table ttest (x int4);




vac=> create trigger tbefore before insert or update or delete on ttest
for each row execute procedure trigf();
CREATE
vac=> create trigger tafter after insert or update or delete on ttest
for each row execute procedure trigf();
CREATE
vac=> insert into ttest values (null);
NOTICE:trigf (fired before): there are 0 tuples in ttest
INSERT 0 0

-- Insertion skipped and AFTER trigger is not fired

vac=> select * from ttest;
x
-
(0 rows)

vac=> insert into ttest values (1);
NOTICE:trigf (fired before): there are 0 tuples in ttest
NOTICE:trigf (fired after ): there are 1 tuples in ttest
                                       ^^^^^^^^
                             remember what we said about visibility.
INSERT 167793 1
vac=> select * from ttest;
x
-
1
(1 row)

vac=> insert into ttest select x * 2 from ttest;
NOTICE:trigf (fired before): there are 1 tuples in ttest
NOTICE:trigf (fired after ): there are 2 tuples in ttest
                                       ^^^^^^^^
                             remember what we said about visibility.
INSERT 167794 1
vac=> select * from ttest;
x
-
1
2
```

```
(2 rows)

vac=> update ttest set x = null where x = 2;
NOTICE:trigf (fired before): there are 2 tuples in ttest
UPDATE 0
vac=> update ttest set x = 4 where x = 2;
NOTICE:trigf (fired before): there are 2 tuples in ttest
NOTICE:trigf (fired after ): there are 2 tuples in ttest
UPDATE 1
vac=> select * from ttest;
x
-
1
4
(2 rows)

vac=> delete from ttest;
NOTICE:trigf (fired before): there are 2 tuples in ttest
NOTICE:trigf (fired after ): there are 1 tuples in ttest
NOTICE:trigf (fired before): there are 1 tuples in ttest
NOTICE:trigf (fired after ): there are 0 tuples in ttest
                                       ^^^^^^^^
                             remember what we said about visibility.
DELETE 2
vac=> select * from ttest;
x
-
(0 rows)
```

Chapter 22. 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.

Procedures which use SPI are called by the Executor. The SPI calls recursively invoke the Executor in turn to run queries. When the Executor is invoked recursively, it may itself call procedures which may make SPI calls.

Note, that if during execution of a query from a procedure the transaction is aborted then control will not be returned to your procedure. Rather, all work will be rolled back and the server will wait for the next command from the client. This will 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.

22.1. 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.

If your procedure is already connected, `SPI_connect` will return an SPI_ERROR_CONNECT error. Note that this may happen if a procedure which has called `SPI_connect` directly calls another procedure which itself calls `SPI_connect`. While recursive calls to the SPI manager are permitted when an SPI query invokes another function which uses SPI, directly nested calls to `SPI_connect` and `SPI_finish` are forbidden.

Usage

Algorithm

`SPI_connect` performs the following:

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

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 arround, 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.

22.2. 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 that 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, fname)
```

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_repalloc

Name

`SPI_repalloc` Re-allocates memory in upper Executor context

Synopsis

`SPI_repalloc(`*`pointer`*`, `*`size`*`)`

Inputs

void * *`pointer`*

Pointer to existing storage

Size *`size`*

Octet size of storage to allocate

Outputs

void *

New storage space of specified size with contents copied from existing area

Description

`SPI_repalloc` re-allocates memory in upper Executor context. 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

22.3. 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 that is chosen as the 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!

22.4. 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's scan. In effect, this duplicates the database table within itself (subject to unique index rules, of course) without recursing.

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

22.5. Examples

This example of SPI usage demonstrates the visibility rule. There are more complex examples in src/test/regress/regress.c and in contrib/spi.

This is a very simple example of SPI usage. The procedure execq accepts an SQL-query in its first argument and tcount in its second, executes the query using SPI_exec and returns the number of tuples for which the query executed:

```
#include "executor/spi.h"   /* this is what you need to work with SPI */

int execq(text *sql, int cnt);

int
execq(text *sql, int cnt)
{
    char *query;
    int ret;
    int proc;

    /* Convert given TEXT object to a C string */
    query = DatumGetCString(DirectFunctionCall1(textout,
                                                PointerGetDatum(sql)));

    SPI_connect();

    ret = SPI_exec(query, cnt);

    proc = SPI_processed;
    /*
     * If this is SELECT and some tuple(s) fetched -
     * returns tuples to the caller via elog (NOTICE).
     */
    if ( ret == SPI_OK_SELECT && SPI_processed > 0 )
    {
        TupleDesc tupdesc = SPI_tuptable->tupdesc;
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        int i,j;

        for (j = 0; j < proc; j++)
        {
            HeapTuple tuple = tuptable->vals[j];

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                sprintf(buf + strlen (buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog (NOTICE, "EXECQ: %s", buf);
        }
    }

    SPI_finish();

    pfree(query);

    return (proc);
}
```

Now, compile and create the function:

```
create function execq (text, 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
```

III. Procedural Languages

Chapter 23. Procedural Languages

Postgres allows users to add new programming languages to be available for writing functions and procedures. These are called *procedural languages* (PL). In the case of a function or trigger procedure written in a procedural language, the database server has no built-in knowledge about how to interpret the function's source text. Instead, the task is passed to a special handler that knows the details of the language. The handler could either do all the work of parsing, syntax analysis, execution, etc. itself, or it could serve as glue between Postgres and an existing implementation of a programming language. The handler itself is a special programming language function compiled into a shared object and loaded on demand.

Writing a handler for a new procedural language is outside the scope of this manual, although some information is provided in the CREATE LANGUAGE reference page. Several procedural languages are available in the standard Postgres distribution.

23.1. Installing Procedural Languages

A procedural language must be installed into each database where it is to be used. But procedural languages installed in the template1 database are automatically available in all subsequently created databases. So the database administrator can decide which languages are available in which databases, and can make some languages available by default if he chooses.

For the languages supplied with the standard distribution, the shell script `createlang` may be used instead of carrying out the details by hand. For example, to install PL/pgSQL into the template1 database, use

```
createlang plpgsql template1
```

The manual procedure described below is only recommended for installing custom languages that `createlang` does not know about.

Manual Procedural Language Installation

A procedural language is installed in the database in three steps, which must be carried out by a database superuser.

1. The shared object for the language handler must be compiled and installed into an appropriate library directory. This works in the same way as building and installing modules with regular user-defined C functions does; see Section 13.4.6.
2. The handler must be declared with the command

   ```
   CREATE FUNCTION handler_function_name ()
       RETURNS OPAQUE AS
       'path-to-shared-object' LANGUAGE 'C';
   ```

   The special return type of `OPAQUE` tells the database that this function does not return one of the defined SQL data types and is not directly usable in SQL statements.

3. The PL must be declared with the command

   ```
   CREATE [TRUSTED] [PROCEDURAL] LANGUAGE 'language-name'
       HANDLER handler_function_name
       LANCOMPILER 'description';
   ```

   The optional key word TRUSTED tells whether ordinary database users that have no superuser privileges should be allowed to use this language to create functions and trigger procedures. Since PL functions are executed inside the database backend, the TRUSTED flag should only be given for languages that do not allow access to database backends internals or the filesystem. The languages PL/pgSQL, PL/Tcl, and PL/Perl are known to be trusted; the language PL/TclU should *not* be marked trusted.

In a default Postgres installation, the handler for the PL/pgSQL language is built and installed into the library directory. If Tcl/Tk support is configured in, the handlers for PL/Tcl and PL/TclU are also built and installed in the same location. Likewise, the PL/Perl handler is built and installed if Perl support is configured. The `createlang` script automates the two CREATE steps described above.

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'.

Chapter 24. PL/pgSQL - SQL Procedural Language

PL/pgSQL is a loadable procedural language for the Postgres database system.

This package was originally written by Jan Wieck. This documentation was in part written by Roberto Mello (<`rmello@fslc.usu.edu`>).

24.1. Overview

The design goals of PL/pgSQL were to create a loadable procedural language that

- can be used to create functions and trigger procedures,
- adds control structures to the SQL language,
- can perform complex computations,
- inherits all user defined types, functions and operators,
- can be defined to be trusted by the server,
- is easy to use.

The PL/pgSQL call handler parses the function's source text and produces an internal binary instruction tree the first time the function is called. The produced bytecode is identified in the call handler by the object ID of the function. This ensures that changing a function by a DROP/CREATE sequence will take effect without establishing a new database connection.

For all expressions and SQL statements used in the function, the PL/pgSQL bytecode interpreter creates a prepared execution plan using the SPI manager's `SPI_prepare()` and `SPI_saveplan()` functions. This is done the first time the individual statement is processed in the PL/pgSQL function. Thus, a function with conditional code that contains many statements for which execution plans would be required, will only prepare and save those plans that are really used during the lifetime of the database connection.

This means that you have to be careful about your user-defined functions. For example:

```
CREATE FUNCTION populate() RETURNS INTEGER AS '
DECLARE
    -- Declarations
BEGIN
    PERFORM my_function();
END;
' LANGUAGE 'plpgsql';
```

If you create the above function, it will reference the OID for `my_function()` in its bytecode. Later, if you drop and re-create `my_function()`, then `populate()` will not be able to find `my_function()` anymore. You would then have to re-create `populate()`.

Because PL/pgSQL saves execution plans in this way, queries that appear directly in a PL/pgSQL function must refer to the same tables and fields on every execution; that is, you cannot use a parameter as the name of a table or field in a query. To get around this restriction, you can construct dynamic queries using the PL/pgSQL EXECUTE statement --- at the price of constructing a new query plan on every execution.

Except for input/output conversion and calculation functions for user defined types, anything that can be defined in C language functions can also be done with PL/pgSQL. It is possible to create complex conditional computation functions and later use them to define operators or use them in functional indices.

24.1.1. Advantages of Using PL/pgSQL

Better performance (see Section 24.1.1.1)

SQL support (see Section 24.1.1.2)

Portability (see Section 24.1.1.3)

24.1.1.1. Better Performance

SQL is the language PostgreSQL (and most other Relational Databases) use as query language. It’s portable and easy to learn. But every SQL statement must be executed individually by the database server.

That means that your client application must send each query to the database server, wait for it to process it, receive the results, do some computation, then send other queries to the server. All this incurs inter process communication and may also incur network overhead if your client is on a different machine than the database server.

With PL/pgSQL you can group a block of computation and a series of queries *inside* the database server, thus having the power of a procedural language and the ease of use of SQL, but saving lots of time because you don’t have the whole client/server communication overhead. Your application will enjoy a considerable performance increase by using PL/pgSQL.

24.1.1.2. SQL Support

PL/pgSQL adds the power of a procedural language to the flexibility and ease of SQL. With PL/pgSQL you can use all the datatypes, columns, operators and functions of SQL.

24.1.1.3. Portability

Because PL/pgSQL functions run inside PostgreSQL, these functions will run on any platform where PostgreSQL runs. Thus you can reuse code and have less development costs.

24.1.2. Developing in PL/pgSQL

Developing in PL/pgSQL is pretty straight forward, especially if you have developed in other database procedural languages, such as Oracle’s PL/SQL. Two good ways of developing in PL/pgSQL are:

Using a text editor and reloading the file with **psql**

Using PostgreSQL’s GUI Tool: pgaccess

One good way to develop in PL/pgSQL is to simply use the text editor of your choice to create your functions, and in another console, use **psql** (PostgreSQL's interactive monitor) to load those functions. If you are doing it this way (and if you are a PL/pgSQL novice or in debugging stage), it is a good idea to always **DROP** your function before creating it. That way when you reload the file, it'll drop your functions and then re-create them. For example:

```
drop function testfunc(integer);
create function testfunc(integer) return integer as ’
    ....
end;
’ language ’plpgsql’;
```

When you load the file for the first time, PostgreSQL will raise a warning saying this function doesn't exist and go on to create it. To load an SQL file (filename.sql) into a database named "dbname", use the command:

```
psql -f filename.sql dbname
```

Another good way to develop in PL/pgSQL is using PostgreSQL's GUI tool: pgaccess. It does some nice things for you, like escaping single-quotes, and making it easy to recreate and debug functions.

24.2. Description

24.2.1. Structure of PL/pgSQL

PL/pgSQL is a *block structured* language. All keywords and identifiers can be used in mixed upper and lower-case. A block is defined as:

```
[<<label>>]
[DECLARE
    declarations]
BEGIN
    statements
END;
```

There can be any number of sub-blocks in the statement section of a block. Sub-blocks can be used to hide variables from outside a block of statements.

The variables declared in the declarations section preceding a block are initialized to their default values every time the block is entered, not only once per function call. For example:

```
CREATE FUNCTION somefunc() RETURNS INTEGER AS ’
DECLARE
   quantity INTEGER := 30;
BEGIN
```

```
   RAISE NOTICE ''Quantity here is %'',quantity;  -- Quantity here is 30
   quantity := 50;
   --
   -- Create a sub-block
   --
   DECLARE
      quantity INTEGER := 80;
   BEGIN
      RAISE NOTICE ''Quantity here is %'',quantity;  -- Quantity here is 80
   END;

   RAISE NOTICE ''Quantity here is %'',quantity;  -- Quantity here is 50
END;
' LANGUAGE 'plpgsql';
```

It is important not to confuse the use of BEGIN/END for grouping statements in PL/pgSQL with the database commands for transaction control. PL/pgSQL's BEGIN/END are only for grouping; they do not start or end a transaction. Functions and trigger procedures are always executed within a transaction established by an outer query --- they cannot start or commit transactions, since Postgres does not have nested transactions.

24.2.2. Comments

There are two types of comments in PL/pgSQL. A double dash `--` starts a comment that extends to the end of the line. A `/*` starts a block comment that extends to the next occurrence of `*/`. Block comments cannot be nested, but double dash comments can be enclosed into a block comment and a double dash can hide the block comment delimiters `/*` and `*/`.

24.2.3. Variables and Constants

All variables, rows and records used in a block or its sub-blocks must be declared in the declarations section of a block. The exception being the loop variable of a FOR loop iterating over a range of integer values.

PL/pgSQL variables can have any SQL datatype, such as `INTEGER`, `VARCHAR` and `CHAR`. All variables have as default value the SQL NULL value.

Here are some examples of variable declarations:

```
user_id INTEGER;
quantity NUMBER(5);
url VARCHAR;
```

24.2.3.1. Constants and Variables With Default Values

The declarations have the following syntax:

```
name [ CONSTANT ] type [ NOT NULL ] [ { DEFAULT | := } value ];
```

The value of variables declared as CONSTANT cannot be changed. If NOT NULL is specified, an assignment of a NULL value results in a runtime error. Since the default value of all variables is the SQL NULL value, all variables declared as NOT NULL must also have a default value specified.

The default value is evaluated every time the function is called. So assigning 'now' to a variable of type `timestamp` causes the variable to have the time of the actual function call, not when the function was precompiled into its bytecode.

Examples:

```
quantity INTEGER := 32;
url varchar := ''http://mysite.com'';
user_id CONSTANT INTEGER := 10;
```

24.2.3.2. Variables Passed to Functions

Variables passed to functions are named with the identifiers `$1`, `$2`, etc. (maximum is 16). Some examples:

```
CREATE FUNCTION sales_tax(REAL) RETURNS REAL AS '
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    return subtotal * 0.06;
END;
' LANGUAGE 'plpgsql';


CREATE FUNCTION instr(VARCHAR,INTEGER) RETURNS INTEGER AS '
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- Some computations here
END;
' LANGUAGE 'plpgsql';
```

24.2.3.3. Attributes

Using the `%TYPE` and `%ROWTYPE` attributes, you can declare variables with the same datatype or structure of another database item (e.g: a table field).

%TYPE

`%TYPE` provides the datatype of a variable or database column. You can use this to declare variables that will hold database values. For example, let's say you have a column named `user_id` in your `users` table. To declare a variable with the same datatype as users you do:

```
user_id users.user_id%TYPE;
```

By using `%TYPE` you don't need to know the datatype of the structure you are referencing, and most important, if the datatype of the referenced item changes in the future (e.g: you change your table definition of user_id to become a REAL), you won't need to change your function definition.

`name` *`table`*%ROWTYPE;

Declares a row with the structure of the given table. *`table`* must be an existing table or view name of the database. The fields of the row are accessed in the dot notation. Parameters to a function can be composite types (complete table rows). In that case, the corresponding identifier $n will be a rowtype, but it must be aliased using the ALIAS command described above.

Only the user attributes of a table row are accessible in the row, no OID or other system attributes (because the row could be from a view). The fields of the rowtype inherit the table's field sizes or precision for `char()` etc. data types.

```
DECLARE
    users_rec users%ROWTYPE;
  user_id users%TYPE;
BEGIN
    user_id := users_rec.user_id;
    ...

create function cs_refresh_one_mv(integer) returns integer as '
   DECLARE
        key ALIAS FOR $1;
        table_data cs_materialized_views%ROWTYPE;
   BEGIN
        SELECT INTO table_data * FROM cs_materialized_views
               WHERE sort_key=key;

        IF NOT FOUND THEN
           RAISE EXCEPTION ''View '' || key || '' not found'';
           RETURN 0;
        END IF;

        -- The mv_name column of cs_materialized_views stores view
        -- names.

        TRUNCATE TABLE table_data.mv_name;
        INSERT INTO table_data.mv_name || '' '' || table_data.mv_query;

        return 1;
end;
' LANGUAGE 'plpgsql';
```

24.2.3.4. RENAME

Using RENAME you can change the name of a variable, record or row. This is useful if NEW or OLD should be referenced by another name inside a trigger procedure.

Syntax and examples:

```
RENAME oldname TO newname;

RENAME id TO user_id;
RENAME this_var TO that_var;
```

24.2.4. Expressions

All expressions used in PL/pgSQL statements are processed using the backend's executor. Expressions that appear to contain constants may in fact require run-time evaluation (e.g. `'now'` for the `timestamp` type) so it is impossible for the PL/pgSQL parser to identify real constant values other than the NULL keyword. All expressions are evaluated internally by executing a query

```
SELECT expression
```

using the SPI manager. In the expression, occurrences of variable identifiers are substituted by parameters and the actual values from the variables are passed to the executor in the parameter array. All expressions used in a PL/pgSQL function are only prepared and saved once. The only exception to this rule is an EXECUTE statement if parsing of a query is needed each time it is encountered.

The type checking done by the Postgres main parser has some side effects to the interpretation of constant values. In detail there is a difference between what these two functions do:

```
CREATE FUNCTION logfunc1 (text) RETURNS timestamp AS '
    DECLARE
        logtxt ALIAS FOR $1;
    BEGIN
        INSERT INTO logtable VALUES (logtxt, ''now'');
        RETURN ''now'';
    END;
' LANGUAGE 'plpgsql';
```

and

```
CREATE FUNCTION logfunc2 (text) RETURNS timestamp AS '
    DECLARE
        logtxt ALIAS FOR $1;
        curtime timestamp;
    BEGIN
        curtime := ''now'';
        INSERT INTO logtable VALUES (logtxt, curtime);
        RETURN curtime;
    END;
' LANGUAGE 'plpgsql';
```

In the case of `logfunc1()`, the Postgres main parser knows when preparing the plan for the INSERT, that the string `'now'` should be interpreted as `timestamp` because the target field of logtable is of that type. Thus, it will make a constant from it at this time and this constant value is then used in all invocations of `logfunc1()` during the lifetime of the backend. Needless to say that this isn't what the programmer wanted.

In the case of `logfunc2()`, the Postgres main parser does not know what type `'now'` should become and therefore it returns a data type of `text` containing the string `'now'`. During the assignment to the local variable curtime, the PL/pgSQL interpreter casts this string to the timestamp type by calling the `text_out()` and `timestamp_in()` functions for the conversion.

This type checking done by the Postgres main parser got implemented after PL/pgSQL was nearly done. It is a difference between 6.3 and 6.4 and affects all functions using the prepared plan feature of the SPI manager. Using a local variable in the above manner is currently the only way in PL/pgSQL to get those values interpreted correctly.

If record fields are used in expressions or statements, the data types of fields should not change between calls of one and the same expression. Keep this in mind when writing trigger procedures that handle events for more than one table.

24.2.5. Statements

Anything not understood by the PL/pgSQL parser as specified below will be put into a query and sent down to the database engine to execute. The resulting query should not return any data.

24.2.5.1. Assignment

An assignment of a value to a variable or row/record field is written as:

```
identifier := expression;
```

If the expressions result data type doesn't match the variables data type, or the variable has a size/precision that is known (as for `char(20)`), the result value will be implicitly casted by the PL/pgSQL bytecode interpreter using the result types output- and the variables type input-functions. Note that this could potentially result in runtime errors generated by the types input functions.

```
user_id := 20;
tax := subtotal * 0.06;
```

24.2.5.2. Calling another function

All functions defined in a Postgres database return a value. Thus, the normal way to call a function is to execute a SELECT query or doing an assignment (resulting in a PL/pgSQL internal SELECT).

But there are cases where someone is not interested in the function's result. In these cases, use the PERFORM statement.

```
PERFORM query
```

This executes a `SELECT` *`query`* over the SPI manager and discards the result. Identifiers like local variables are still substituted into parameters.

```
PERFORM create_mv(''cs_session_page_requests_mv'',''
     select   session_id, page_id, count(*) as n_hits,
              sum(dwell_time) as dwell_time, count(dwell_time) as dwell_count
     from     cs_fact_table
     group by session_id, page_id ’’);
```

24.2.5.3. Executing dynamic queries

Often times you will want to generate dynamic queries inside your PL/pgSQL functions. Or you have functions that will generate other functions. PL/pgSQL provides the EXECUTE statement for these occasions.

```
EXECUTE query-string
```

where *`query-string`* is a string of type `text` containing the *`query`* to be executed.

When working with dynamic queries you will have to face escaping of single quotes in PL/pgSQL. Please refer to the table available at the "Porting from Oracle PL/SQL" chapter for a detailed explanation that will save you some effort.

Unlike all other queries in PL/pgSQL, a *`query`* run by an EXECUTE statement is not prepared and saved just once during the life of the server. Instead, the *`query`* is prepared each time the statement is run. The *`query-string`* can be dynamically created within the procedure to perform actions on variable tables and fields.

The results from SELECT queries are discarded by EXECUTE, and SELECT INTO is not currently supported within EXECUTE. So, the only way to extract a result from a dynamically-created SELECT is to use the FOR ... EXECUTE form described later.

An example:

```
EXECUTE ’’UPDATE tbl SET ’’
        || quote_ident(fieldname)
        || ’’ = ’’
        || quote_literal(newvalue)
        || ’’ WHERE ...’’;
```

This example shows use of the functions `quote_ident(TEXT)` and `quote_literal(TEXT)`. Variables containing field and table identifiers should be passed to function `quote_ident()`. Variables containing literal elements of the dynamic query string should be passed to `quote_literal()`. Both take the appropriate steps to return the input text enclosed in single or double quotes and with any embedded special characters.

Here is a much larger example of a dynamic query and EXECUTE:

```
CREATE FUNCTION cs_update_referrer_type_proc() RETURNS INTEGER AS ’
DECLARE
    referrer_keys RECORD;  -- Declare a generic record to be used in a FOR
    a_output varchar(4000);
BEGIN
               a_output            :=            ’’CREATE            FUNCTION
cs_find_referrer_type(varchar,varchar,varchar)
                  RETURNS varchar AS ’’’’
                     DECLARE
                         v_host ALIAS FOR $1;
                         v_domain ALIAS FOR $2;
                         v_url ALIAS FOR $3; ’’;
    --
    -- Notice how we scan through the results of a query in a FOR loop
    -- using the FOR <record> construct.
    --

    FOR referrer_keys IN select * from cs_referrer_keys order by try_order
LOOP
        a_output := a_output || '' if v_'' || referrer_keys.kind || '' like
''''''''''
                 || referrer_keys.key_string || '''''''''' then return ''''''
                 || referrer_keys.referrer_type || ''''''; end if;'';
    END LOOP;

     a_output := a_output || '' return null; end; '''' language
''''plpgsql'''';'';

    -- This works because we are not substituting any variables
     -- Otherwise it would fail. Look at PERFORM for another way to run
functions

    EXECUTE a_output;
end;
' LANGUAGE 'plpgsql';
```

24.2.5.4. Obtaining other results status

GET DIAGNOSTICS *variable* = *item* [, ...]

This command allows retrieval of system status indicators. Each *item* is a keyword identifying a state value to be assigned to the specified variable (which should be of the right datatype to receive it). The currently available status items are `ROW_COUNT`, the number of rows processed by the last SQL query sent down to the SQL engine; and `RESULT_OID`, the Oid of the last row inserted by the most recent SQL query. Note that `RESULT_OID` is only useful after an INSERT query.

24.2.5.5. Returning from a function

RETURN *expression*

The function terminates and the value of *expression* will be returned to the upper executor. The return value of a function cannot be undefined. If control reaches the end of the top-level block of the function without hitting a RETURN statement, a runtime error will occur.

The expressions result will be automatically casted into the function's return type as described for assignments.

24.2.6. Control Structures

Control structures are probably the most useful (and important) part of PL/SQL. With PL/pgSQL's control structures, you can manipulate PostgreSQL data in a very flexible and powerful way.

24.2.6.1. Conditional Control: IF statements

`IF` statements let you take action according to certain conditions. PL/pgSQL has three forms of IF: IF-THEN, IF-THEN-ELSE, IF-THEN-ELSE IF. NOTE: All PL/pgSQL IF statements need a corresponding `END IF` statement. In ELSE-IF statements you need two: one for the first IF and one for the second (ELSE IF).

IF-THEN

IF-THEN statements is the simplest form of an IF. The statements between THEN and END IF will be executed if the condition is true. Otherwise, the statements following END IF will be executed.

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

IF-THEN-ELSE

IF-THEN-ELSE statements adds to IF-THEN by letting you specify the statements that should be executed if the condition evaluates to FALSE.

```
IF parentid IS NULL or parentid = ''''
THEN
    return fullname;
ELSE
    return hp_true_filename(parentid) || ''/'' || fullname;
END IF;
```

```
IF v_count > 0 THEN
    INSERT INTO users_count(count) VALUES(v_count);
    return ''t'';
ELSE
    return ''f'';
END IF;
```

IF statements can be nested and in the following example:

```
IF demo_row.sex = ''m'' THEN
  pretty_sex := ''man'';
ELSE
  IF demo_row.sex = ''f'' THEN
    pretty_sex := ''woman'';
  END IF;
END IF;
```

IF-THEN-ELSE IF

When you use the "ELSE IF" statement, you are actually nesting an IF statement inside the ELSE statement. Thus you need one END IF statement for each nested IF and one for the parent IF-ELSE.

For example:

```
IF demo_row.sex = ''m'' THEN
   pretty_sex := ''man'';
ELSE IF demo_row.sex = ''f'' THEN
        pretty_sex := ''woman'';
     END IF;
END IF;
```

24.2.6.2. Iterative Control: LOOP, WHILE, FOR and EXIT

With the LOOP, WHILE, FOR and EXIT statements, you can control the flow of execution of your PL/pgSQL program iteratively.

LOOP

```
[<<label>>]
LOOP
    statements
END LOOP;
```

An unconditional loop that must be terminated explicitly by an EXIT statement. The optional label can be used by EXIT statements of nested loops to specify which level of nesting should be terminated.

EXIT

```
EXIT [ label ] [ WHEN expression ];
```

If no *label* is given, the innermost loop is terminated and the statement following END LOOP is executed next. If *label* is given, it must be the label of the current or an upper level of nested loop blocks. Then the named loop or block is terminated and control continues with the statement after the loops/blocks corresponding END.

Examples:

```
LOOP
    -- some computations
    IF count > 0 THEN
        EXIT;  -- exit loop
    END IF;
END LOOP;
```

```
LOOP
    -- some computations
    EXIT WHEN count > 0;
END LOOP;

BEGIN
    -- some computations
    IF stocks > 100000 THEN
        EXIT;  -- illegal. Can’t use EXIT outside of a LOOP
    END IF;
END;
```

WHILE

With the WHILE statement, you can loop through a sequence of statements as long as the evaluation of the condition expression is true.

```
[<<label>>]
WHILE expression LOOP
    statements
END LOOP;
```

For example:

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- some computations here
END LOOP;

WHILE NOT boolean_expression LOOP
    -- some computations here
END LOOP;
```

FOR

```
[<<label>>]
FOR name IN [ REVERSE ] expression .. expression LOOP
    statements
END LOOP;
```

A loop that iterates over a range of integer values. The variable *`name`* is automatically created as type integer and exists only inside the loop. The two expressions giving the lower and upper bound of the range are evaluated only when entering the loop. The iteration step is always 1.

Some examples of FOR loops (see Section 24.2.7 for iterating over records in FOR loops):

```
FOR i IN 1..10 LOOP
  -- some expressions here

    RAISE NOTICE ’i is %’,i;
END LOOP;
```

```
FOR i IN REVERSE 1..10 LOOP
    -- some expressions here
END LOOP;
```

24.2.7. Working with RECORDs

Records are similar to rowtypes, but they have no predefined structure. They are used in selections and FOR loops to hold one actual database row from a SELECT operation.

24.2.7.1. Declaration

One variables of type RECORD can be used for different selections. Accessing a record or an attempt to assign a value to a record field when there is no actual row in it results in a runtime error. They can be declared like this:

```
name RECORD;
```

24.2.7.2. Assignments

An assignment of a complete selection into a record or row can be done by:

```
SELECT  INTO target expressions FROM ...;
```

`target` can be a record, a row variable or a comma separated list of variables and record-/row-fields. Note that this is quite different from Postgres' normal interpretation of SELECT INTO, which is that the INTO target is a newly created table. (If you want to create a table from a SELECT result inside a PL/pgSQL function, use the equivalent syntax **CREATE TABLE AS SELECT**.)

If a row or a variable list is used as target, the selected values must exactly match the structure of the target(s) or a runtime error occurs. The FROM keyword can be followed by any valid qualification, grouping, sorting etc. that can be given for a SELECT statement.

Once a record or row has been assigned to a RECORD variable, you can use the "." (dot) notation to access fields in that record:

```
DECLARE
    users_rec RECORD;
    full_name varchar;
BEGIN
    SELECT INTO users_rec * FROM users WHERE user_id=3;

  full_name := users_rec.first_name || '' '' || users_rec.last_name;
```

There is a special variable named FOUND of type `boolean` that can be used immediately after a SELECT INTO to check if an assignment had success.

```
SELECT INTO myrec * FROM EMP WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION ''employee % not found'', myname;
END IF;
```

You can also use the IS NULL (or ISNULL) conditionals to test for NULLity of a RECORD/ROW. If the selection returns multiple rows, only the first is moved into the target fields. All others are silently discarded.

```
DECLARE
    users_rec RECORD;
    full_name varchar;
BEGIN
    SELECT INTO users_rec * FROM users WHERE user_id=3;

    IF users_rec.homepage IS NULL THEN
        -- user entered no homepage, return "http://"

        return ''http://'';
    END IF;
END;
```

24.2.7.3. Iterating Through Records

Using a special type of FOR loop, you can iterate through the results of a query and manipulate that data accordingly. The syntax is as follow:

```
[<<label>>]
FOR record | row IN select_clause LOOP
    statements
END LOOP;
```

The record or row is assigned all the rows resulting from the select clause and the loop body executed for each. Here is an example:

```
create function cs_refresh_mviews () returns integer as '
DECLARE
     mviews RECORD;

     -- Instead, if you did:
     -- mviews  cs_materialized_views%ROWTYPE;
     -- this record would ONLY be usable for the cs_materialized_views table

BEGIN
     PERFORM cs_log(''Refreshing materialized views...'');

     FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP

         -- Now "mviews" has one record from cs_materialized_views
```

```
            PERFORM cs_log(''Refreshing materialized view ''
             || mview.mv_name || ''...'');
            TRUNCATE TABLE mview.mv_name;
            INSERT INTO mview.mv_name || '' '' || mview.mv_query;
        END LOOP;

        PERFORM cs_log(''Done refreshing materialized views.'');
        return 1;
end;
' language 'plpgsql';
```

If the loop is terminated with an EXIT statement, the last assigned row is still accessible after the loop.

The FOR-IN EXECUTE statement is another way to iterate over records:

```
[<<label>>]
FOR record | row IN EXECUTE text_expression LOOP
    statements
END LOOP;
```

This is like the previous form, except that the source SELECT statement is specified as a string expression, which is evaluated and re-planned on each entry to the FOR loop. This allows the programmer to choose the speed of a pre-planned query or the flexibility of a dynamic query, just as with a plain EXECUTE statement.

24.2.8. Aborting and Messages

Use the RAISE statement to throw messages into the Postgres elog mechanism.

```
RAISE level 'format' [, identifier [...]];
```

Inside the format, `%` is used as a placeholder for the subsequent comma-separated identifiers. Possible levels are DEBUG (silently suppressed in production running databases), NOTICE (written into the database log and forwarded to the client application) and EXCEPTION (written into the database log and aborting the transaction).

```
RAISE NOTICE ''Id number '' || key || '' not found!'';
RAISE NOTICE ''Calling cs_create_job(%)'',v_job_id;
```

In this last example, v_job_id will replace the % in the string.

```
RAISE EXCEPTION ''Inexistent ID --> %'',user_id;
```

This will abort the transaction and write to the database log.

24.2.9. Exceptions

Postgres does not have a very smart exception handling model. Whenever the parser, planner/optimizer or executor decide that a statement cannot be processed any longer, the whole transaction gets aborted and the system jumps back into the main loop to get the next query from the client application.

It is possible to hook into the error mechanism to notice that this happens. But currently it is impossible to tell what really caused the abort (input/output conversion error, floating point error, parse error). And it is possible that the database backend is in an inconsistent state at this point so returning to the upper executor or issuing more commands might corrupt the whole database. And even if, at this point the information, that the transaction is aborted, is already sent to the client application, so resuming operation does not make any sense.

Thus, the only thing PL/pgSQL currently does when it encounters an abort during execution of a function or trigger procedure is to write some additional DEBUG level log messages telling in which function and where (line number and type of statement) this happened.

24.3. Trigger Procedures

PL/pgSQL can be used to define trigger procedures. They are created with the usual **CREATE FUNCTION** command as a function with no arguments and a return type of `OPAQUE`.

There are some Postgres specific details in functions used as trigger procedures.

First they have some special variables created automatically in the top-level blocks declaration section. They are

`NEW`

Data type `RECORD`; variable holding the new database row on INSERT/UPDATE operations on ROW level triggers.

`OLD`

Data type `RECORD`; variable holding the old database row on UPDATE/DELETE operations on ROW level triggers.

`TG_NAME`

Data type `name`; variable that contains the name of the trigger actually fired.

`TG_WHEN`

Data type `text`; a string of either `BEFORE` or `AFTER` depending on the triggers definition.

`TG_LEVEL`

Data type `text`; a string of either `ROW` or `STATEMENT` depending on the triggers definition.

`TG_OP`

Data type `text`; a string of `INSERT`, `UPDATE` or `DELETE` telling for which operation the trigger is actually fired.

`TG_RELID`

Data type `oid`; the object ID of the table that caused the trigger invocation.

`TG_RELNAME`

Data type `name`; the name of the table that caused the trigger invocation.

`TG_NARGS`

Data type `integer`; the number of arguments given to the trigger procedure in the **CREATE TRIGGER** statement.

`TG_ARGV[]`

Data type array of `text`; the arguments from the **CREATE TRIGGER** statement. The index counts from 0 and can be given as an expression. Invalid indices (< 0 or >= tg_nargs) result in a NULL value.

Second they must return either NULL or a record/row containing exactly the structure of the table the trigger was fired for. Triggers fired AFTER might always return a NULL value with no effect. Triggers fired BEFORE signal the trigger manager to skip the operation for this actual row when returning NULL. Otherwise, the returned record/row replaces the inserted/updated row in the operation. It is possible to replace single values directly in NEW and return that or to build a complete new record/row to return.

Example 24-1. A PL/pgSQL Trigger Procedure Example

This trigger ensures, that any time a row is inserted or updated in the table, the current user name and time are stamped into the row. And it ensures that an employees name is given and that the salary is a positive value.

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp () RETURNS OPAQUE AS ’
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname ISNULL THEN
            RAISE EXCEPTION ’’empname cannot be NULL value’’;
        END IF;
        IF NEW.salary ISNULL THEN
            RAISE EXCEPTION ’’% cannot have NULL salary’’, NEW.empname;
        END IF;

        -- Who works for us when she must pay for?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION ’’% cannot have a negative salary’’, NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := ’’now’’;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
’ LANGUAGE ’plpgsql’;
```

```
CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

24.4. Examples

Here are only a few functions to demonstrate how easy it is to write PL/pgSQL functions. For more complex examples the programmer might look at the regression test for PL/pgSQL.

One painful detail in writing functions in PL/pgSQL is the handling of single quotes. The function's source text on **CREATE FUNCTION** must be a literal string. Single quotes inside of literal strings must be either doubled or quoted with a backslash. We are still looking for an elegant alternative. In the meantime, doubling the single quotes as in the examples below should be used. Any solution for this in future versions of Postgres will be forward compatible.

For a detailed explanation and examples of how to escape single quotes in different situations, please see Section 24.5.1.1.

Example 24-2. A Simple PL/pgSQL Function to Increment an Integer

The following two PL/pgSQL functions are identical to their counterparts from the C language function discussion. This function receives an `integer` and increments it by one, returning the incremented value.

```
CREATE FUNCTION add_one (integer) RETURNS integer AS '
    BEGIN
        RETURN $1 + 1;
    END;
' LANGUAGE 'plpgsql';
```

Example 24-3. A Simple PL/pgSQL Function to Concatenate Text

This function receives two `text` parameters and returns the result of concatenating them.

```
CREATE FUNCTION concat_text (text, text) RETURNS text AS '
    BEGIN
        RETURN $1 || $2;
    END;
' LANGUAGE 'plpgsql';
```

Example 24-4. A PL/pgSQL Function on Composite Type

In this example, we take EMP (a table) and an `integer` as arguments to our function, which returns a `boolean`. If the "salary" field of the EMP table is `NULL`, we return "f". Otherwise we compare with that field with the `integer` passed to the function and return the `boolean` result of the comparison (t or f). This is the PL/pgSQL equivalent to the example from the C functions.

```
CREATE FUNCTION c_overpaid (EMP, integer) RETURNS boolean AS '
    DECLARE
        emprec ALIAS FOR $1;
        sallim ALIAS FOR $2;
    BEGIN
        IF emprec.salary ISNULL THEN
            RETURN ''f'';
        END IF;
        RETURN emprec.salary > sallim;
    END;
' LANGUAGE 'plpgsql';
```

24.5. Porting from Oracle PL/SQL

Author: Roberto Mello (<rmello@fslc.usu.edu>)

This section explains differences between Oracle's PL/SQL and PostgreSQL's PL/pgSQL languages in the hopes of helping developers port applications from Oracle to PostgreSQL. Most of the code here is from the ArsDigita (http://www.arsdigita.com) Clickstream module (http://www.arsdigita.com/asj/clickstream) that I ported to PostgreSQL when I took an internship with OpenForce Inc. (http://www.openforce.net) in the Summer of 2000.

PL/pgSQL is similar to PL/SQL in many aspects. It is a block structured, imperative language (all variables have to be declared). PL/SQL has many more features than its PostgreSQL counterpart, but PL/pgSQL allows for a great deal of functionality and it is being improved constantly.

24.5.1. Main Differences

Some things you should keep in mind when porting from Oracle to PostgreSQL:

No default parameters in PostgreSQL.

You can overload functions in PostgreSQL. This is often used to work around the lack of default parameters.

Assignments, loops and conditionals are similar.

No need for cursors in PostgreSQL, just put the query in the FOR statement (see example below)

In PostgreSQL you *need* to escape single quotes. See Section 24.5.1.1.

24.5.1.1. Quote Me on That: Escaping Single Quotes

In PostgreSQL you need to escape single quotes inside your function definition. This can lead to quite amusing code at times, especially if you are creating a function that generates other function(s), as in Example 24-6. One thing to keep in mind when escaping lots of single quotes is that, except for the beginning/ending quotes, all the others will come in even quantity.

Table 24-1 gives the scoop. (You'll love this little chart.)

Table 24-1. Single Quotes Escaping Chart

No. of Quotes	Usage	Example	Result
1	To begin/terminate function bodies	`CREATE FUNCTION foo() RETURNS INTEGER AS '...' LANGUAGE 'plpgsql';`	as is

No. of Quotes	Usage	Example	Result
2	In assignments, SELECTs, to delimit strings, etc.	`a_output := ’’Blah’’; SELECT * FROM users WHERE f_name=’’foobar’’;`	`SELECT * FROM users WHERE f_name=’foobar’;`
4	When you need two single quotes in your resulting string without terminating that string.	`a_output := a_output || ’’ AND name LIKE ’’’’foobar’’’’ AND ...’’`	`AND name LIKE ’foobar’ AND ...`
6	When you want double quotes in your resulting string *and* terminate that string.	`a_output := a_output || ’’ AND name LIKE ’’’’foobar’’’’’’`	`AND name LIKE ’foobar’`
10	When you want two single quotes in the resulting string (which accounts for 8 quotes) *and* terminate that string (2 more). You will probably only need that if you were using a function to generate other functions (like in Example 24-6).	`a_output := a_output || ’’ if v_’’ || referrer_keys.kind || ’’ like ’’’’’’’’’’ || referrer_keys.key_string || ’’’’’’’’’’ then return ’’’’’’ || referrer_keys.referrer_ty- pe || ’’’’’’; end if;’’;`	`if v_<...> like ’’<...>’’ then return ’’<...>’’; end if;`

24.5.2. Porting Functions

Example 24-5. A Simple Function

Here is an Oracle function:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name IN varchar,
v_version IN varchar)
RETURN varchar IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || ’/’ || v_version;
END;
/
SHOW ERRORS;
```

Let’s go through this function and see the differences to PL/pgSQL:

The `OR REPLACE` clause is not allowed. You will have to explicitly drop the function before creating it to achieve similar results.

PostgreSQL does not have named parameters. You have to explicitly alias them inside your function.

Oracle can have `IN`, `OUT`, and `INOUT` parameters passed to functions. The `INOUT`, for example, means that the parameter will receive a value and return another. PostgreSQL only has `IN` parameters and functions can return only a single value.

The `RETURN` key word in the function prototype (not the function body) becomes `RETURNS` in PostgreSQL.

On PostgreSQL functions are created using single quotes as delimiters, so you have to escape single quotes inside your functions (which can be quite annoying at times; see Section 24.5.1.1).

The `/show errors` command does not exist in PostgreSQL.

So let’s see how this function would be look like ported to PostgreSQL:

```
DROP FUNCTION cs_fmt_browser_version(varchar, varchar);
CREATE FUNCTION cs_fmt_browser_version(varchar, varchar)
RETRUNS varchar AS ’
DECLARE
    v_name ALIAS FOR $1;
    v_version ALIAS FOR $2;
BEGIN
    IF v_version IS NULL THEN
        return v_name;
    END IF;
    RETURN v_name || ’’/’’ || v_version;
END;
’ LANGUAGE ’plpgsql’;
```

Example 24-6. A Function that Creates Another Function

The following procedure grabs rows from a `SELECT` statement and builds a large function with the results in `IF` statements, for the sake of efficiency. Notice particularly the differences in cursors, `FOR` loops, and the need to escape single quotes in PostgreSQL.

```
create or replace procedure cs_update_referrer_type_proc is
    cursor referrer_keys is
        select * from cs_referrer_keys
        order by try_order;

    a_output varchar(4000);
begin
    a_output := ’create or replace function cs_find_referrer_type(v_host IN
varchar, v_domain IN varchar,
v_url IN varchar) return varchar is begin’;

    for referrer_key in referrer_keys loop
```

```
        a_output := a_output || ' if v_' || referrer_key.kind || ' like '''
||
referrer_key.key_string || ''' then return ''' || referrer_key.referrer_type
||
'''; end if;';
    end loop;

    a_output := a_output || ' return null; end;';
    execute immediate a_output;
end;
/
show errors
```

Here is how this function would end up in PostgreSQL:

```
CREATE FUNCTION cs_update_referrer_type_proc() RETURNS integer AS '
DECLARE
    referrer_keys RECORD;  -- Declare a generic record to be used in a FOR
    a_output varchar(4000);
BEGIN
               a_output            :=            ''CREATE            FUNCTION
cs_find_referrer_type(varchar,varchar,varchar)
                  RETURNS varchar AS ''''
                     DECLARE
                         v_host ALIAS FOR $1;
                         v_domain ALIAS FOR $2;
                         v_url ALIAS FOR $3; '';

    --
    -- Notice how we scan through the results of a query in a FOR loop
    -- using the FOR <record> construct.
    --

     FOR referrer_keys IN select * from cs_referrer_keys order by try_order
LOOP
        a_output := a_output || '' if v_'' || referrer_keys.kind || '' like
''''''''''
                 || referrer_keys.key_string || '''''''''' then return ''''''
                 || referrer_keys.referrer_type || ''''''; end if;'';
    END LOOP;

     a_output  :=  a_output  ||  ''  return  null;  end;  ''''  language
''''plpgsql'''';'';

    -- This works because we are not substituting any variables
     -- Otherwise it would fail. Look at PERFORM for another way to run
functions

    EXECUTE a_output;
end;
' LANGUAGE 'plpgsql';
```

Example 24-7. A Procedure with a lot of String Manipulation and OUT Parameters

The following Oracle PL/SQL procedure is used to parse a URL and return several elements (host, path and query). It is an procedure because in PL/pgSQL functions only one value can be returned (see Section 24.5.3). In PostgreSQL, one way to work around this is to split the procedure in three different functions: one to return the host, another for the path and another for the query.

```
create or replace procedure cs_parse_url(
    v_url IN varchar,
    v_host OUT varchar,  -- This will be passed back
    v_path OUT varchar,  -- This one too
    v_query OUT varchar) -- And this one
is
    a_pos1 integer;
    a_pos2 integer;
begin
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
     a_pos1  :=  instr(v_url,  ’//’);  --  PostgreSQL  doesn’t  have  an  instr
function

    if a_pos1 = 0 then
        return;
    end if;
    a_pos2 := instr(v_url, ’/’, a_pos1 + 2);
    if a_pos2 = 0 then
        v_host := substr(v_url, a_pos1 + 2);
        v_path := ’/’;
        return;
    end if;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, ’?’, a_pos2 + 1);

    if a_pos1 = 0 then
        v_path := substr(v_url, a_pos2);
        return;
    end if;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
end;
/
show errors;
```

Here is how this procedure could be translated for PostgreSQL:

```
drop function cs_parse_url_host(varchar);
create function cs_parse_url_host(varchar) returns varchar as ’
declare
    v_url ALIAS FOR $1;
    v_host varchar;
    v_path varchar;
    a_pos1 integer;
    a_pos2 integer;
    a_pos3 integer;
begin
    v_host := NULL;
    a_pos1 := instr(v_url,''//'');

    if a_pos1 = 0 then
        return '''';  -- Return a blank
    end if;

    a_pos2 := instr(v_url,''/'',a_pos1 + 2);
    if a_pos2 = 0 then
        v_host := substr(v_url, a_pos1 + 2);
        v_path := ''/'';
        return v_host;
    end if;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2 );
    return v_host;
end;
' language 'plpgsql';
```

Note: PostgreSQL does not have an `instr` function, so you can work around it using a combination of other functions. I got tired of doing this and created my own `instr` functions that behave exactly like Oracle's (it makes life easier). See the Section 24.5.6 for the code.

24.5.3. Procedures

Oracle procedures give a little more flexibility to the developer because nothing needs to be explicitly returned, but it can be through the use of INOUT or OUT parameters.

An example:

```
create or replace procedure cs_create_job(v_job_id in integer)
is
    a_running_job_count integer;
    pragma autonomous_transaction;
begin
    lock table cs_jobs in exclusive mode;

    select count(*) into a_running_job_count from cs_jobs
    where end_stamp is null;

    if a_running_job_count > 0 then
        commit; -- free lock
        raise_application_error(-20000, 'Unable to create a new job: a job is
currently running.');
    end if;

    delete from cs_active_job;
    insert into cs_active_job(job_id) values(v_job_id);
    begin
        insert into cs_jobs(job_id, start_stamp) values(v_job_id, sysdate);
         exception when dup_val_on_index then null; -- don’t worry if it
already exists
    end;
    commit;
end;
/
show errors
```

Procedures like this can be easily converted into PostgreSQL functions returning an `INTEGER`. This procedure in particular is interesting because it can teach us some things:

- There is no `pragma` statement in PostgreSQL.
- If you do a `LOCK TABLE` in PL/pgSQL, the lock will not be released until the calling transaction is finished.
- You also cannot have transactions in PL/pgSQL procedures. The entire function (and other functions called from therein) is executed in a transaction and PostgreSQL rolls back the results if something goes wrong. Therefore only one `BEGIN` statement is allowed.
- The exception when would have to be replaced by an `IF` statement.

So let’s see one of the ways we could port this procedure to PL/pgSQL:

```
drop function cs_create_job(integer);
create function cs_create_job(integer) returns integer as ’ declare
    v_job_id alias for $1;
    a_running_job_count integer;
    a_num integer;
    -- pragma autonomous_transaction;
begin
    lock table cs_jobs in exclusive mode;
    select count(*) into a_running_job_count from cs_jobs where end_stamp is
null;

    if a_running_job_count > 0 then
        -- commit; -- free lock
         raise exception ’’Unable to create a new job: a job is currently
running.’’;
    end if;

    delete from cs_active_job;
    insert into cs_active_job(job_id) values(v_job_id);

    SELECT count(*) into a_num FROM cs_jobs WHERE job_id=v_job_id;
    IF NOT FOUND THEN  -- If nothing was returned in the last query
        -- This job is not in the table so lets insert it.
        insert into cs_jobs(job_id, start_stamp) values(v_job_id, sysdate());
        return 1;
    ELSE
        raise NOTICE ''Job already running.'';
    END IF;

    return 0;
end;
' language 'plpgsql';
```

Notice how you can raise notices (or errors) in PL/pgSQL.

24.5.4. Packages

Note: I haven't done much with packages myself, so if there are mistakes here, please let me know.

Packages are a way Oracle gives you to encapsulate PL/SQL statements and functions into one entity, like Java classes, where you define methods and objects. You can access these objects/methods with a `.` (dot). Here is an example of an Oracle package from ACS 4 (the ArsDigita Community System (http://www.arsdigita.com/doc/)):

```
create or replace package body acs
as
  function add_user (
    user_id     in users.user_id%TYPE default null,
    object_type     in acs_objects.object_type%TYPE
               default 'user',
    creation_date   in acs_objects.creation_date%TYPE
               default sysdate,
    creation_user   in acs_objects.creation_user%TYPE
               default null,
    creation_ip     in acs_objects.creation_ip%TYPE default null,
  ...
  ) return users.user_id%TYPE
  is
    v_user_id       users.user_id%TYPE;
    v_rel_id        membership_rels.rel_id%TYPE;
  begin
    v_user_id := acs_user.new (user_id, object_type, creation_date,
                creation_user, creation_ip, email,
    ...
    return v_user_id;
  end;
end acs;
/
show errors
```

We port this to PostgreSQL by creating the different objects of the Oracle package as functions with a standard naming convention. We have to pay attention to some other details, like the lack of default parameters in PostgreSQL functions. The above package would become something like this:

```
CREATE FUNCTION
acs__add_user(integer,integer,varchar,datetime,integer,integer,...)
RETURNS integer AS ’
DECLARE
    user_id ALIAS FOR $1;
    object_type ALIAS FOR $2;
    creation_date ALIAS FOR $3;
    creation_user ALIAS FOR $4;
    creation_ip ALIAS FOR $5;
    ...
    v_user_id users.user_id%TYPE;
    v_rel_id membership_rels.rel_id%TYPE;
BEGIN
    v_user_id :=
acs_user__new(user_id,object_type,creation_date,creation_user,creation_ip,
...);
    ...

    return v_user_id;
END;
’ LANGUAGE ’plpgsql’;
```

24.5.5. Other Things to Watch For

24.5.5.1. EXECUTE

The PostgreSQL version of `EXECUTE` works nicely, but you have to remember to use `quote_literal(TEXT)` and `quote_string(TEXT)` as described in Section 24.2.5.3. Constructs of the type `EXECUTE ’’SELECT * from $1’’;` will not work unless you use these functions.

24.5.5.2. Optimizing PL/pgSQL Functions

PostgreSQL gives you two function creation modifiers to optimize execution: `iscachable` (function always returns the same result when given the same arguments) and `isstrict` (function returns NULL if any argument is NULL). Consult the **CREATE FUNCTION** reference for details.

To make use of these optimization attributes, you have to use the `WITH` modifier in your **CREATE FUNCTION** statement. Something like:

```
CREATE FUNCTION foo(...) RETURNS integer AS ’
...
’ LANGUAGE ’plpgsql’
WITH (isstrict, iscachable);
```

24.5.6. Appendix

24.5.6.1. Code for my `instr` functions

* *This function should probably be integrated into the core.*

```
--
-- instr functions that mimic Oracle’s counterpart
-- Syntax: instr(string1,string2,[n],[m]) where [] denotes optional params.
--
-- Searches string1 beginning at the nth character for the mth
-- occurrence of string2. If n is negative, search backwards. If m is
-- not passed, assume 1 (search starts at first character).
--
-- by Roberto Mello (rmello@fslc.usu.edu)
-- modified by Robert Gaszewski (graszew@poland.com)
-- Licensed under the GPL v2 or later.
--

DROP FUNCTION instr(varchar,varchar);
CREATE FUNCTION instr(varchar,varchar) RETURNS integer AS ’
DECLARE
    pos integer;
BEGIN
    pos:= instr($1,$2,1);
    RETURN pos;
END;
’ language ’plpgsql’;


DROP FUNCTION instr(varchar,varchar,integer);
CREATE FUNCTION instr(varchar,varchar,integer) RETURNS integer AS ’
DECLARE
    string ALIAS FOR $1;
    string_to_search ALIAS FOR $2;
    beg_index ALIAS FOR $3;
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN

       temp_str := substring(string FROM beg_index);
       pos := position(string_to_search IN temp_str);

       IF pos = 0 THEN
                 RETURN 0;
             ELSE
                 RETURN pos + beg_index - 1;
```

```
                END IF;
    ELSE
       ss_length := char_length(string_to_search);
       length := char_length(string);
       beg := length + beg_index - ss_length + 2;

       WHILE beg > 0 LOOP

           temp_str := substring(string FROM beg FOR ss_length);
                 pos := position(string_to_search IN temp_str);

                 IF pos > 0 THEN
                           RETURN beg;
                 END IF;

                 beg := beg - 1;
       END LOOP;
       RETURN 0;
    END IF;
END;
' language 'plpgsql';

--
-- Written by Robert Gaszewski (graszew@poland.com)
-- Licensed under the GPL v2 or later.
--
DROP FUNCTION instr(varchar,varchar,integer,integer);
CREATE FUNCTION instr(varchar,varchar,integer,integer) RETURNS integer AS '
DECLARE
    string ALIAS FOR $1;
    string_to_search ALIAS FOR $2;
    beg_index ALIAS FOR $3;
    occur_index ALIAS FOR $4;
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        beg := beg_index;
        temp_str := substring(string FROM beg_index);

        FOR i IN 1..occur_index LOOP
            pos := position(string_to_search IN temp_str);

            IF i = 1 THEN
                beg := beg + pos - 1;
            ELSE
                beg := beg + pos;
            END IF;
            temp_str := substring(string FROM beg + 1);
        END LOOP;

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN beg;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                occur_number := occur_number + 1;

                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    END IF;
END;
' language 'plpgsql';
```

Chapter 25. PL/Tcl - TCL Procedural Language

PL/Tcl is a loadable procedural language for the Postgres database system that enables the Tcl language to be used to create functions and trigger procedures.

This package was originally written by Jan Wieck.

25.1. Overview

PL/Tcl offers most of the capabilities a function writer has in the C language, except for some restrictions.

The good restriction is that everything is executed in a safe Tcl interpreter. In addition to the limited command set of safe Tcl, only a few commands are available to access the database via SPI and to raise messages via elog(). There is no way to access internals of the database backend or to gain OS-level access under the permissions of the Postgres user ID, as a C function can do. Thus, any unprivileged database user may be permitted to use this language.

The other, implementation restriction is that Tcl procedures cannot be used to create input/output functions for new data types.

Sometimes it is desirable to write Tcl functions that are not restricted to safe Tcl --- for example, one might want a Tcl function that sends mail. To handle these cases, there is a variant of PL/Tcl called PL/TclU (for untrusted Tcl). This is the exact same language except that a full Tcl interpreter is used. *If PL/TclU is used, it must be installed as an untrusted procedural language* so that only database superusers can create functions in it. The writer of a PL/TclU function must take care that the function cannot be used to do anything unwanted, since it will be able to do anything that could be done by a user logged in as the database administrator.

The shared object for the PL/Tcl and PL/TclU call handlers is automatically built and installed in the Postgres library directory if Tcl/Tk support is specified in the configuration step of the installation procedure. To install PL/Tcl and/or PL/TclU in a particular database, use the `createlang` script.

25.2. Description

25.2.1. Postgres Functions and Tcl Procedure Names

In Postgres, one and the same function name can be used for different functions as long as the number of arguments or their types differ. This would collide with Tcl procedure names. To offer the same flexibility in PL/Tcl, the internal Tcl procedure names contain the object ID of the procedure's pg_proc row as part of their name. Thus, different argtype versions of the same Postgres function are different for Tcl too.

25.2.2. Defining Functions in PL/Tcl

To create a function in the PL/Tcl language, use the standard syntax

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS '
```

```
    # PL/Tcl function body
' LANGUAGE 'pltcl';
```

When the function is called, the arguments are given as variables $1 ... $n to the Tcl procedure body. For example, a function returning the higher of two int4 values could be defined as:

```
CREATE FUNCTION tcl_max (int4, int4) RETURNS int4 AS '
    if {$1 > $2} {return $1}
    return $2
' LANGUAGE 'pltcl';
```

Composite type arguments are given to the procedure as Tcl arrays. The element names in the array are the attribute names of the composite type. If an attribute in the actual row has the NULL value, it will not appear in the array! Here is an example that defines the overpaid_2 function (as found in the older Postgres documentation) in PL/Tcl

```
CREATE FUNCTION overpaid_2 (EMP) RETURNS bool AS '
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
' LANGUAGE 'pltcl';
```

25.2.3. Global Data in PL/Tcl

Sometimes (especially when using the SPI functions described later) it is useful to have some global status data that is held between two calls to a procedure. This is easily done since all PL/Tcl procedures executed in one backend share the same safe Tcl interpreter.

To help protect PL/Tcl procedures from unwanted side effects, an array is made available to each procedure via the upvar command. The global name of this variable is the procedure's internal name and the local name is GD. It is recommended that GD be used for private status data of a procedure. Use regular Tcl global variables only for values that you specifically intend to be shared among multiple procedures.

25.2.4. Trigger Procedures in PL/Tcl

Trigger procedures are defined in Postgres as functions without arguments and a return type of opaque. And so are they in the PL/Tcl language.

The information from the trigger manager is given to the procedure body in the following variables:

$TG_name

The name of the trigger from the CREATE TRIGGER statement.

$TG_relid

The object ID of the table that caused the trigger procedure to be invoked.

$TG_relatts

A Tcl list of the tables field names prefixed with an empty list element. So looking up an element name in the list with the lsearch Tcl command returns the same positive number starting from 1 as the fields are numbered in the pg_attribute system catalog.

$TG_when

The string BEFORE or AFTER depending on the event of the trigger call.

$TG_level

The string ROW or STATEMENT depending on the event of the trigger call.

$TG_op

The string INSERT, UPDATE or DELETE depending on the event of the trigger call.

$NEW

An array containing the values of the new table row on INSERT/UPDATE actions, or empty on DELETE.

$OLD

An array containing the values of the old table row on UPDATE/DELETE actions, or empty on INSERT.

$GD

The global status data array as described above.

$args

A Tcl list of the arguments to the procedure as given in the CREATE TRIGGER statement. The arguments are also accessible as $1 ... $n in the procedure body.

The return value from a trigger procedure is one of the strings OK or SKIP, or a list as returned by the ’array get’ Tcl command. If the return value is OK, the normal operation (INSERT/UPDATE/DELETE) that fired this trigger will take place. Obviously, SKIP tells the trigger manager to silently suppress the operation. The list from ’array get’ tells PL/Tcl to return a modified row to the trigger manager that will be inserted instead of the one given in $NEW (INSERT/UPDATE only). Needless to say that all this is only meaningful when the trigger is BEFORE and FOR EACH ROW.

Here’s a little example trigger procedure that forces an integer value in a table to keep track of the number of updates that are performed on the row. For new rows inserted, the value is initialized to 0 and then incremented on every update operation:

```
CREATE FUNCTION trigfunc_modcount() RETURNS OPAQUE AS ’
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
' LANGUAGE 'pltcl';

CREATE TABLE mytab (num int4, modcnt int4, description text);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

25.2.5. Database Access from PL/Tcl

The following commands are available to access the database from the body of a PL/Tcl procedure:

elog *level msg*

Fire a log message. Possible levels are NOTICE, ERROR, FATAL, DEBUG and NOIND as for the `elog` C function.

quote *string*

Duplicates all occurrences of single quote and backslash characters. It should be used when variables are used in the query string given to `spi_exec` or `spi_prepare` (not for the value list on `spi_execp`). Think about a query string like

```
"SELECT '$val' AS ret"
```

where the Tcl variable val actually contains "doesn't". This would result in the final query string

```
"SELECT 'doesn't' AS ret"
```

which would cause a parse error during `spi_exec` or `spi_prepare`. It should contain

```
"SELECT 'doesn''t' AS ret"
```

and has to be written as

```
"SELECT '[ quote $val ]' AS ret"
```

spi_exec ?-count *n*? ?-array *name*? *query* ?*loop-body*?

Call parser/planner/optimizer/executor for query. The optional -count value tells `spi_exec` the maximum number of rows to be processed by the query.

If the query is a SELECT statement and the optional loop-body (a body of Tcl commands like in a foreach statement) is given, it is evaluated for each row selected and behaves like expected on continue/break. The values of selected fields are put into variables named as the column names. So a

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

will set the variable $cnt to the number of rows in the pg_proc system catalog. If the option -array is given, the column values are stored in the associative array named ’name’ indexed by the column name instead of individual variables.

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

will print a DEBUG log message for every row of pg_class. The return value of `spi_exec` is the number of rows affected by the query as found in the global variable SPI_processed.

spi_prepare *query typelist*

Prepares AND SAVES a query plan for later execution. It is a bit different from the C level SPI_prepare in that the plan is automatically copied to the toplevel memory context. Thus, there is currently no way of preparing a plan without saving it.

If the query references arguments, the type names must be given as a Tcl list. The return value from spi_prepare is a query ID to be used in subsequent calls to spi_execp. See spi_execp for a sample.

spi_exec ?-count *n*? ?-array*name*? ?-nulls*string*? *queryid* ?*value-list*? ?*loop-body*?

Execute a prepared plan from spi_prepare with variable substitution. The optional -count value tells spi_execp the maximum number of rows to be processed by the query.

The optional value for -nulls is a string of spaces and ’n’ characters telling spi_execp which of the values are NULL’s. If given, it must have exactly the length of the number of values.

The queryid is the ID returned by the spi_prepare call.

If there was a typelist given to spi_prepare, a Tcl list of values of exactly the same length must be given to spi_execp after the query. If the type list on spi_prepare was empty, this argument must be omitted.

If the query is a SELECT statement, the same as described for spi_exec happens for the loop-body and the variables for the fields selected.

Here’s an example for a PL/Tcl function using a prepared plan:

```
CREATE FUNCTION t1_count(int4, int4) RETURNS int4 AS ’
    if {![ info exists GD(plan) ]} {
        # prepare the saved plan on the first call
        set GD(plan) [ spi_prepare \\
                "SELECT count(*) AS cnt FROM t1 WHERE num >= \\$1 AND num
<= \\$2" \\
                int4 ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
' LANGUAGE 'pltcl';
```

Note that each backslash that Tcl should see must be doubled in the query creating the function, since the main parser processes backslashes too on CREATE FUNCTION. Inside the query string given to spi_prepare should really be dollar signs to mark the parameter positions and to not let $1 be substituted by the value given in the first function call.

Modules and the unknown command

PL/Tcl has a special support for things often used. It recognizes two magic tables, pltcl_modules and pltcl_modfuncs. If these exist, the module 'unknown' is loaded into the interpreter right after creation. Whenever an unknown Tcl procedure is called, the unknown proc is asked to check if the procedure is defined in one of the modules. If this is true, the module is loaded on demand. To enable this behavior, the PL/Tcl call handler must be compiled with -DPLTCL_UNKNOWN_SUPPORT set.

There are support scripts to maintain these tables in the modules subdirectory of the PL/Tcl source including the source for the unknown module that must get installed initially.

Chapter 26. PL/Perl - Perl Procedural Language

PL/Perl allows you to write functions in the Perl programming language that may be used in SQL queries as if they were built into Postgres.

The PL/Perl intepreter is a full Perl interpreter. However, certain operations have been disabled in order to maintain the security of the system. In general, the operations that are restricted are those that interact with the environment. This includes filehandle operations, `require`, and `use` (for external modules). It should be noted that this security is not absolute. Indeed, several Denial-of-Service attacks are still possible - memory exhaustion and endless loops are two examples.

26.1. Building and Installing

In order to build and install PL/Perl if you are installing Postgres from source then the `--with-perl` must be supplied to the `configure` script. PL/Perl requires that, when Perl was installed, the `libperl` library was build as a shared object. At the time of this writing, this is almost never the case in the Perl packages that are distributed with the operating systems. A message like this will appear during the build to point out this fact:

```
*****
* Cannot build PL/Perl because libperl is not a shared library.
* Skipped.
*****
```

Therefore it is likely that you will have to re-build and install Perl manually to be able to build PL/Perl.

When you want to retry to build PL/Perl after having reinstalled Perl, then change to the directory `src/pl/plperl` in the Postgres source tree and issue the commands

```
gmake clean
gmake all
gmake install
```

The **createlang** command is used to install the language into a database.

```
$ createlang plperl template1
```

If it is installed into template1, all future databases will have the language installed automatically.

26.2. Using PL/Perl

Assume you have the following table:

```
CREATE TABLE EMPLOYEE (
    name text,
    basesalary integer,
    bonus integer
);
```

In order to get the total compensation (base + bonus) we could define a function as follows:

```
CREATE FUNCTION totalcomp(integer, integer) RETURNS integer
    AS ’return $_[0] + $_[1]’
    LANGUAGE ’plperl’;
```

Notice that the arguments to the function are passed in `@_` as might be expected.

We can now use our function like so:

```
SELECT name, totalcomp(basesalary, bonus) FROM employee;
```

But, we can also pass entire tuples to our functions:

```
CREATE FUNCTION empcomp(employee) RETURNS integer AS ’
    my $emp = shift;
    return $emp->{’’basesalary’’} + $emp->{’’bonus’’};
’ LANGUAGE ’plperl’;
```

A tuple is passed as a reference to a hash. The keys are the names of the fields in the tuples. The hash values are values of the corresponding fields in the tuple.

> **Tip:** Because the function body is passed as an SQL string literal to **CREATE FUNCTION** you have to escape single quotes within your Perl source, either by doubling them as shown above, or by using the extended quoting functions (`q[]`, `qq[]`, `qw[]`). Backslashes must be escaped by doubling them.

The new function `empcomp` can used like:

```
SELECT name, empcomp(employee) FROM employee;
```

Here is an example of a function that will not work because file system operations are not allowed for security reasons:

```
CREATE FUNCTION badfunc() RETURNS integer AS ’
    open(TEMP, ">/tmp/badfile");
    print TEMP "Gotcha!\n";
    return 1;
’ LANGUAGE ’plperl’;
```

The creation of the function will succeed, but executing it will not.

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, Berkeley, Computer Science Department web site (http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/)

SQL Reference Books

The Practical SQL Handbook , Bowman et al, 1996 , *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 , Date and Darwen, 1997 , *A user's guide to the standard database language SQL* , 4, C. J. Date and Hugh Darwen, 0-201-96426-0, 1997, Addison-Wesley, 1997.

An Introduction to Database Systems , Date, 1994 , 6, C. J. Date, 1, 1994, Addison-Wesley, 1994.

Understanding the New SQL , Melton and Simon, 1993 , *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 , Ullman, 1988 , Jeffrey D. Ullman, 1, Computer Science Press , 1988 .

PostgreSQL-Specific Documentation

The PostgreSQL Administrator's Guide , The Administrator's Guide , Edited by Thomas Lockhart, 2001-04-13, The PostgreSQL Global Development Group.

The PostgreSQL Developer's Guide , The Developer's Guide , Edited by Thomas Lockhart, 2001-04-13, The PostgreSQL Global Development Group.

The PostgreSQL Programmer's Guide , The Programmer's Guide , Edited by Thomas Lockhart, 2001-04-13, The PostgreSQL Global Development Group.

The PostgreSQL Tutorial Introduction , The Tutorial , Edited by Thomas Lockhart, 2001-04-13, The PostgreSQL Global Development Group.

The PostgreSQL User's Guide , The User's Guide , Edited by Thomas Lockhart, 2001-04-13, The PostgreSQL Global Development Group.

Enhancement of the ANSI SQL Implementation of PostgreSQL , Simkovics, 1998 , Stefan Simkovics, 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 , Yu and Chen, 1995 , 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 , Olson, 1993 , 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 , Ong and Goh, 1990 , 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 , L. Rowe and M. Stonebraker, Sept. 1987, VLDB Conference, Brighton, England, 1987.

Generalized partial indexes (http://simon.cs.cornell.edu/home/praveen/papers/partindex.de95.ps.Z) , Seshardri, 1995 , 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 , Stonebraker and Rowe, 1986 , M. Stonebraker and L. Rowe, May 1986, Conference on Management of Data, Washington DC, ACM-SIGMOD, 1986.

The Design of the Postgres Rules System, Stonebraker, Hanson, Hong, 1987 , M. Stonebraker, E. Hanson, and C. H. Hong, Feb. 1987, Conference on Data Engineering, Los Angeles, CA, IEEE, 1987.

The Postgres Storage System , Stonebraker, 1987 , M. Stonebraker, Sept. 1987, VLDB Conference, Brighton, England, 1987.

A Commentary on the Postgres Rules System , Stonebraker et al, 1989, 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) , Stonebraker, M, 1989b, M. Stonebraker, Dec. 1989, Record 18(no.4):4-11, SIGMOD, 1989.

The Implementation of Postgres , Stonebraker, Rowe, Hirohama, 1990 , 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 , Stonebraker et al, ACM, 1990 , M. Stonebraker and et al, June 1990, Conference on Management of Data, ACM-SIGMOD.