PostgreSQL 8.4.4 Documentation

The PostgreSQL Global Development Group

PostgreSQL 8.4.4 Documentation

The PostgreSQL Global Development Group

Legal Notice

Table of Contents

Preface

This book is the official documentation of PostgreSQL™. It has been written by the PostgreSQL™ developers and other volunteers in parallel to the development of the PostgreSQL™ software. It describes all the functionality that the current version of PostgreSQL™ officially supports.

To make the large amount of information about PostgreSQL™ manageable, this book has been organized in several parts. Each part is targeted at a different class of users, or at users in different stages of their PostgreSQL™ experience:

- Part I, “Tutorial” is an informal introduction for new users.
- ??? documents the SQL query language environment, including data types and functions, as well as user-level performance tuning. Every PostgreSQL™ user should read this.
- ??? describes the installation and administration of the server. Everyone who runs a PostgreSQL™ server, be it for private use or for others, should read this part.
- ??? describes the programming interfaces for PostgreSQL™ client programs.
- ??? contains information for advanced users about the extensibility capabilities of the server. Topics include user-defined data types and functions.
- ??? contains reference information about SQL commands, client and server programs. This part supports the other parts with structured information sorted by command or program.
- ??? contains assorted information that might be of use to PostgreSQL™ developers.

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. POSTGRES pioneered many concepts that only became available in some commercial database systems much later.

PostgreSQL™ is an open-source descendant of this original Berkeley code. It supports a large part of the SQL standard and offers many modern features:

- complex queries
- foreign keys
- triggers
- views
- transactional integrity
- multiversion concurrency control

Also, PostgreSQL™ can be extended by the user in many ways, for example by adding new

- data types
- functions
- operators
- aggregate functions
- index methods
- procedural languages

And because of the liberal license, PostgreSQL™ can be used, modified, and distributed by anyone free of charge for any purpose, be it private, commercial, or academic.

A Brief History of PostgreSQL™

The object-relational database management system now known as PostgreSQL™ is derived from the POSTGRES™ package written at the University of California at Berkeley. With over two decades of development behind it, PostgreSQL™ is now the most advanced open-source database available anywhere.

The Berkeley POSTGRES™ Project

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. The implementation of POSTGRES™ began in 1986. The initial concepts for the system were presented in [STON86], and the definition of the initial data model appeared in [ROWE87]. The design of the rule system at that time was described in [STON87a]. The rationale and architecture of the storage manager were detailed in [STON87b].

POSTGRES™ has undergone several major releases since then. The first "demoware" system became operational in 1987 and was shown at the 1988 ACM-SIGMOD Conference. Version 1, described in [STON90a], was released to a few external users in June 1989. In response to a critique of the first rule system ([STON89]), the rule system was redesigned ([STON90b]), and Version 2 was released in June 1990 with the new rule system. Version 3 appeared in 1991 and added support for multiple storage managers, an improved query executor, and a rewritten rule system. For the most part, subsequent releases until Postgres95™ (see below) focused on portability and reliability.

POSTGRES™ has been used to implement many different research and production applications. These include: a financial data analysis system, a jet engine performance monitoring package, an asteroid tracking database, a medical information database, and several geographic information systems. POSTGRES™ has also been used as an educational tool at several universities. Finally, Illustra Information Technologies (later merged into Informix™ [http://www.informix.com/], which is now owned by IBM [http://www.ibm.com/]) picked up the code and commercialized it. In late 1992, POSTGRES™ became the primary data manager for the Sequoia 2000 scientific computing project [http://meteora.ucsd.edu/s2k/s2k_home.html].

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

Postgres95™

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

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

- The query language PostQUEL was replaced with SQL (implemented in the server). Subqueries were not supported until PostgreSQL™ (see below), but they could be imitated in Postgres95™ with user-

defined SQL functions. Aggregate functions were re-implemented. Support for the GROUP BY query clause was also added.

- A new program (psql) was provided for interactive SQL queries, which used GNU Readline. This largely superseded the old monitor program.
- 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™ server.
- 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).

PostgreSQL™

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

Many people continue to refer to PostgreSQL™ as "Postgres" (now rarely in all capital letters) because of tradition or because it is easier to pronounce. This usage is widely accepted as a nickname or alias.

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

Details about what has happened in PostgreSQL™ since then can be found in ???.

Conventions

This book uses the following typographical conventions to mark certain portions of text: new terms, foreign phrases, and other important passages are emphasized in *italics*. Everything that represents input or output of the computer, in particular commands, program code, and screen output, is shown in a monospaced font (example). Within such passages, italics (*example*) indicate placeholders; you must insert an actual value instead of the placeholder. On occasion, parts of program code are emphasized in bold face (**example**), if they have been added or changed since the preceding example.

The following conventions are used in the synopsis of a command: brackets ([and]) indicate optional parts. (In the synopsis of a Tcl command, question marks (?) are used instead, as is usual in Tcl.) Braces ({ and }) and vertical lines (|) indicate that you must choose one alternative. Dots (...) mean that the preceding element can be repeated.

Where it enhances the clarity, SQL commands are preceded by the prompt =>, and shell commands are preceded by the prompt $. Normally, prompts are not shown, though.

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 book does not have fixed presumptions about system administration procedures.

Further Information

Besides the documentation, that is, this book, there are other resources about PostgreSQL™:

Wiki
: The PostgreSQL™ wiki [http://wiki.postgresql.org] contains the project's FAQ [http://wiki.postgresql.org/wiki/Frequently_Asked_Questions] (Frequently Asked Questions) list, TODO [http://wiki.postgresql.org/wiki/Todo] list, and detailed information about many more topics.

Web Site
: The PostgreSQL™ web site [http://www.postgresql.org] carries details on the latest release and other information to make your work or play with PostgreSQL™ more productive.

Mailing Lists
: The mailing lists are a good place to have your questions answered, to share experiences with other users, and to contact the developers. Consult the PostgreSQL™ web site for details.

Yourself!
: PostgreSQL™ is an open-source project. 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. Read the mailing lists and answer questions. If you learn something which is not in the documentation, write it up and contribute it. If you add features to the code, contribute them.

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 doing so 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.

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 a program does something different from what the documentation says, that is a bug. That might include, but is not limited to, the following circumstances:

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

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

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

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

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 SQL-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 way to create this file is to use pg_dump to dump out the table declarations and data needed to set the scene, then add the problem query. You are encouraged to minimize the size of your example, but this is not absolutely necessary. If the bug is reproducible, we will find it either way.

  If your application uses some other client interface, such as PHP, then please try to isolate the offending queries. We will probably not set up a web server to reproduce your problem. In any case remember to provide the exact input files; do not guess that the problem happens for “large files” or “midsize 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

If you are reporting an error message, please obtain the most verbose form of the message. In psql, say \set VERBOSITY verbose beforehand. If you are extracting the message from the server log, set the run-time parameter ??? to verbose so that all details are logged.

Note

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

- The output you expected is very important to state. If you just write "This command gives me that output." or "This is not what I expected.", we might run it ourselves, scan the output, and think it looks OK and is exactly what we expected. We should not have to spend the time to decode the exact semantics behind your commands. Especially refrain from merely saying that "This is not what SQL says/Oracle does." Digging out the correct behavior from SQL is not a fun undertaking, nor do we all know how all the other relational databases out there behave. (If your problem is a program crash, you can obviously omit this item.)

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

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

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

  If your version is older than 8.4.4 we will almost certainly tell you to upgrade. There are many bug fixes and improvements in each new release, so it is quite possible that a bug you have encountered in an older release of PostgreSQL™ has already been fixed. We can only provide limited support for sites using older releases of PostgreSQL™; if you require more than we can provide, consider acquiring a commercial support contract.

- Platform information. This includes the kernel name and version, C library, processor, memory information, and so on. 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 i386s. If you have installation problems then information about the toolchain on your machine (compiler, make, and so on) 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. Here is an article [http://www.chiark.greenend.org.uk/~sgtatham/bugs.html] that outlines some more tips on reporting bugs.

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

Where to report bugs

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

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

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

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

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

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

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

Note

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

Part I. Tutorial

Welcome to the PostgreSQL™ Tutorial. The following few chapters are intended to give a simple introduction to PostgreSQL™, relational database concepts, and the SQL language to those who are new to any one of these aspects. We only assume some general knowledge about how to use computers. No particular Unix or programming experience is required. This part is mainly intended to give you some hands-on experience with important aspects of the PostgreSQL™ system. It makes no attempt to be a complete or thorough treatment of the topics it covers.

After you have worked through this tutorial you might want to move on to reading ??? to gain a more formal knowledge of the SQL language, or ??? for information about developing applications for PostgreSQL™. Those who set up and manage their own server should also read ???.

Chapter 1. Getting Started

Installation

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

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

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

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

Architectural Fundamentals

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

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

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

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

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

Creating a Database

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

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

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

```
$ createdb mydb
```

If this produces no response then this step was successful and you can skip over the remainder of this section.

If you see a message similar to:

```
createdb: command not found
```

then PostgreSQL™ was not installed properly. Either it was not installed at all or your shell's search path was not set to include it. Try calling the command with an absolute path instead:

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

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

Another response could be this:

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

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

Another response could be this:

```
createdb: could not connect to database postgres: FATAL:  role "joe" does not exist
```

where your own login name is mentioned. This will happen if the administrator has not created a PostgreSQL™ user account for you. (PostgreSQL™ user accounts are distinct from operating system user accounts.) If you are the administrator, see ??? for help creating accounts. You will need to become the operating system user under which PostgreSQL™ was installed (usually postgres) to create the first user account. It could also be that you were assigned a PostgreSQL™ user name that is different from your operating system user name; in that case you need to use the -U switch or set the PGUSER environment variable to specify your PostgreSQL™ user name.

If you have a user account but it does not have the privileges required to create a database, you will see the following:

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

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

You can also create databases with other names. PostgreSQL™ allows you to create any number of databases at a given site. Database names must have an alphabetic first character and are limited to 63 characters in length. A convenient choice is to create a database with the same name as your current user name. Many tools assume that database name as the default, so it can save you some typing. To create that database, simply type:

```
$ createdb
```

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

```
$ dropdb mydb
```

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

More about **createdb** and **dropdb** can be found in ??? and ??? respectively.

Accessing a Database

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

- Running the PostgreSQL™ interactive terminal program, called *psql*, which allows you to interactively enter, edit, and execute SQL commands.
- Using an existing graphical frontend tool like pgAdmin or an office suite with ODBC or JDBC support to create and manipulate a database. These possibilities are not covered in this tutorial.
- Writing a custom application, using one of the several available language bindings. These possibilities are discussed further in ???.

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

```
$ psql mydb
```

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

If you do not supply the database name then it will default to your user account name. You already discovered this scheme in the previous section using **createdb**.

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

```
psql ()
Type "help" for help.

mydb=>
```

The last line could also be:

```
mydb=#
```

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

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

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

```
mydb=> SELECT version();
                          version
 ---------------------------------------------------------------------------
 PostgreSQL  on i586-pc-linux-gnu, compiled by GCC 2.96, 32-bit
(1 row)

mydb=> SELECT current_date;
   date
------------
 2002-08-31
(1 row)

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

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

```
mydb=> \h
```

To get out of **psql**, type:

```
mydb=> \q
```

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

Chapter 2. The SQL Language

Introduction

This chapter provides an overview of how to use SQL to perform simple operations. This tutorial is only intended to give you an introduction and is in no way a complete tutorial on SQL. Numerous books have been written on SQL, including [MELT93] and [DATE97]. You should be aware that some PostgreSQL™ language features are extensions to the standard.

In the examples that follow, we assume that you have created a database named mydb, as described in the previous chapter, and have been able to start psql.

Examples in this manual can also be found in the PostgreSQL™ source distribution in the directory src/tutorial/. To use those files, first change to that directory and run make:

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

This creates the scripts and compiles the C files containing user-defined functions and types. (If you installed a pre-packaged version of PostgreSQL™ rather than building from source, look for a directory named tutorial within the PostgreSQL™ distribution. The “make” part should already have been done for you.) Then, to start the tutorial, do the following:

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

...

mydb=> \i basics.sql
```

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

Concepts

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

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

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

Creating a New Table

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

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

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

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

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

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

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

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

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

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

```
DROP TABLE tablename;
```

Populating a Table With Rows

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Querying a Table

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

```
SELECT * FROM weather;
```

Here `*` is a shorthand for “all columns”. [1] So the same result would be had with:

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

The output should be:

[1] While `SELECT *` is useful for off-the-cuff queries, it is widely considered bad style in production code, since adding a column to the table would change the results.

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

You can write expressions, not just simple column references, in the select list. For example, you can do:

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

This should give:

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

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

A query can be “qualified” by adding a WHERE clause that specifies which rows are wanted. The WHERE clause contains a Boolean (truth value) expression, and only rows for which the Boolean expression is true are returned. The usual Boolean operators (AND, OR, and NOT) are allowed in the qualification. For example, the following retrieves the weather of San Francisco on rainy days:

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

Result:

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

You can request that the results of a query be returned in sorted order:

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

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

In this example, the sort order isn't fully specified, and so you might get the San Francisco rows in either order. But you'd always get the results shown above if you do:

```
SELECT * FROM weather
  ORDER BY city, temp_lo;
```

You can request that duplicate rows be removed from the result of a query:

```
SELECT DISTINCT city
  FROM weather;
```

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

Here again, the result row ordering might vary. You can ensure consistent results by using DISTINCT and ORDER BY together: [2]

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

Joins Between Tables

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

Note

This is only a conceptual model. The join is usually performed in a more efficient manner than actually comparing each possible pair of rows, but this is invisible to the user.

This would be accomplished by the following query:

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

```
    city     | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+-----------
```

[2] In some database systems, including older versions of PostgreSQL™, the implementation of DISTINCT automatically orders the rows and so ORDER BY is unnecessary. But this is not required by the SQL standard, and current PostgreSQL™ does not guarantee that DISTINCT causes the rows to be ordered.

```
 San Francisco |     46 |     50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |     43 |     57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(2 rows)
```

Observe two things about the result set:

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

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

Exercise: Attempt to determine the semantics of this query when the WHERE clause is omitted.

Since the columns all had different names, the parser automatically found which table they belong to. If there were duplicate column names in the two tables you'd need to *qualify* the column names to show which one you meant, as in:

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

It is widely considered good style to qualify all column names in a join query, so that the query won't fail if a duplicate column name is later added to one of the tables.

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

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

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

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

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

    city     | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+-----------
 Hayward      |     37 |     54 |      | 1994-11-29 |               |
```

```
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(3 rows)
```

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

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

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

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

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

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

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

You will encounter this style of abbreviating quite frequently.

Aggregate Functions

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

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

```
SELECT max(temp_lo) FROM weather;

 max
-----
```

```
 46
(1 row)
```

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

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

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

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

```
    city
---------------
 San Francisco
(1 row)
```

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

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

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

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

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

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

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

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

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

❶ The LIKE operator does pattern matching and is explained in ???.

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

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

Updates

You can update existing rows using the **UPDATE** command. Suppose you discover the temperature readings are all off by 2 degrees after November 28. You can correct the data as follows:

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

Look at the new state of the data:

```
SELECT * FROM weather;

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

Deletions

Rows can be removed from a table using the **DELETE** command. Suppose you are no longer interested in the weather of Hayward. Then you can do the following to delete those rows from the table:

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

All weather records belonging to Hayward are removed.

```
SELECT * FROM weather;
```

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

One should be wary of statements of the form

```
DELETE FROM tablename;
```

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

Chapter 3. Advanced Features

Introduction

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

This chapter will on occasion refer to examples found in Chapter 2, *The SQL Language* to change or improve them, so it will be useful to have read that chapter. Some examples from this chapter can also be found in advanced.sql in the tutorial directory. This file also contains some sample data to load, which is not repeated here. (Refer to the section called "Introduction" for how to use the file.)

Views

Refer back to the queries in the section called "Joins Between Tables". Suppose the combined listing of weather records and city location is of particular interest to your application, but you do not want to type the query each time you need it. You can create a *view* over the query, which gives a name to the query that you can refer to like an ordinary table:

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

SELECT * FROM myview;
```

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

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

Foreign Keys

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

The new declaration of the tables would look like this:

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

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

Now try inserting an invalid record:

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

```
ERROR:  insert or update on table "weather" violates foreign key constraint "weather_city_fkey"
DETAIL:  Key (city)=(Berkeley) is not present in table "cities".
```

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

Transactions

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

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

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

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

We also want a guarantee that once a transaction is completed and acknowledged by the database system, it has indeed been permanently recorded and won't be lost even if a crash ensues shortly thereafter. For example, if we are recording a cash withdrawal by Bob, we do not want any chance that the debit to his account will disappear in a crash just after he walks out the bank door. A transactional database guarantees that all the updates made by a transaction are logged in permanent storage (i.e., on disk) before the transaction is reported complete.

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

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

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

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

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

Note

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

It's possible to control the statements in a transaction in a more granular fashion through the use of *savepoints*. Savepoints allow you to selectively discard parts of the transaction, while committing the rest. After defining a savepoint with **SAVEPOINT**, you can if needed roll back to the savepoint with **ROLLBACK TO**. All the transaction's database changes between defining the savepoint and rolling back to it are discarded, but changes earlier than the savepoint are kept.

After rolling back to a savepoint, it continues to be defined, so you can roll back to it several times. Conversely, if you are sure you won't need to roll back to a particular savepoint again, it can be released, so the system can free some resources. Keep in mind that either releasing or rolling back to a savepoint will automatically release all savepoints that were defined after it.

All this is happening within the transaction block, so none of it is visible to other database sessions. When and if you commit the transaction block, the committed actions become visible as a unit to other sessions, while the rolled-back actions never become visible at all.

Remembering the bank database, suppose we debit $100.00 from Alice's account, and credit Bob's account, only to find later that we should have credited Wally's account. We could do it using savepoints like this:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

This example is, of course, oversimplified, but there's a lot of control possible in a transaction block through the use of savepoints. Moreover, **ROLLBACK TO** is the only way to regain control of a transaction block that was put in aborted state by the system due to an error, short of rolling it back completely and starting again.

Window Functions

A *window function* performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result.

Here is an example that shows how to compare each employee's salary with the average salary in his or her department:

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;
```

```
  depname  | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

The first three output columns come directly from the table empsalary, and there is one output row for each row in the table. The fourth column represents an average taken across all the table rows that have the same *depname* value as the current row. (This actually is the same function as the regular avg aggregate

function, but the OVER clause causes it to be treated as a window function and computed across an appropriate set of rows.)

A window function call always contains an OVER clause following the window function's name and argument(s). This is what syntactically distinguishes it from a regular function or aggregate function. The OVER clause determines exactly how the rows of the query are split up for processing by the window function. The PARTITION BY list within OVER specifies dividing the rows into groups, or partitions, that share the same values of the PARTITION BY expression(s). For each row, the window function is computed across the rows that fall into the same partition as the current row.

Although avg will produce the same result no matter what order it processes the partition's rows in, this is not true of all window functions. When needed, you can control that order using ORDER BY within OVER. Here is an example:

```
SELECT depname, empno, salary, rank() OVER (PARTITION BY depname ORDER BY salary DESC) FROM empsalary;
```

```
  depname  | empno | salary | rank
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    10 |   5200 |    2
 develop   |    11 |   5200 |    2
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
 personnel |     2 |   3900 |    1
 personnel |     5 |   3500 |    2
 sales     |     1 |   5000 |    1
 sales     |     4 |   4800 |    2
 sales     |     3 |   4800 |    2
(10 rows)
```

As shown here, the rank function produces a numerical rank within the current row's partition for each distinct ORDER BY value, in the order defined by the ORDER BY clause. rank needs no explicit parameter, because its behavior is entirely determined by the OVER clause.

The rows considered by a window function are those of the “virtual table” produced by the query's FROM clause as filtered by its WHERE, GROUP BY, and HAVING clauses if any. For example, a row removed because it does not meet the WHERE condition is not seen by any window function. A query can contain multiple window functions that slice up the data in different ways by means of different OVER clauses, but they all act on the same collection of rows defined by this virtual table.

We already saw that ORDER BY can be omitted if the ordering of rows is not important. It is also possible to omit PARTITION BY, in which case there is just one partition containing all the rows.

There is another important concept associated with window functions: for each row, there is a set of rows within its partition called its *window frame*. Many (but not all) window functions act only on the rows of the window frame, rather than of the whole partition. By default, if ORDER BY is supplied then the frame consists of all rows from the start of the partition up through the current row, plus any following rows that are equal to the current row according to the ORDER BY clause. When ORDER BY is omitted the default frame consists of all rows in the partition. [1] Here is an example using sum:

[1] There are options to define the window frame in other ways, but this tutorial does not cover them. See ??? for details.

```
SELECT salary, sum(salary) OVER () FROM empsalary;
```

```
 salary |  sum
--------+-------
   5200 | 47100
   5000 | 47100
   3500 | 47100
   4800 | 47100
   3900 | 47100
   4200 | 47100
   4500 | 47100
   4800 | 47100
   6000 | 47100
   5200 | 47100
(10 rows)
```

Above, since there is no ORDER BY in the OVER clause, the window frame is the same as the partition, which for lack of PARTITION BY is the whole table; in other words each sum is taken over the whole table and so we get the same result for each output row. But if we add an ORDER BY clause, we get very different results:

```
SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;
```

```
 salary |  sum
--------+-------
   3500 |  3500
   3900 |  7400
   4200 | 11600
   4500 | 16100
   4800 | 25700
   4800 | 25700
   5000 | 30700
   5200 | 41100
   5200 | 41100
   6000 | 47100
(10 rows)
```

Here the sum is taken from the first (lowest) salary up through the current one, including any duplicates of the current one (notice the results for the duplicated salaries).

Window functions are permitted only in the SELECT list and the ORDER BY clause of the query. They are forbidden elsewhere, such as in GROUP BY, HAVING and WHERE clauses. This is because they logically execute after the processing of those clauses. Also, window functions execute after regular aggregate functions. This means it is valid to include an aggregate function call in the arguments of a window function, but not vice versa.

If there is a need to filter or group rows after the window calculations are performed, you can use a sub-select. For example:

```
SELECT depname, empno, salary, enroll_date
```

```
FROM
  (SELECT depname, empno, salary, enroll_date,
        rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos
    FROM empsalary
  ) AS ss
WHERE pos < 3;
```

The above query only shows the rows from the inner query having rank less than 3.

When a query involves multiple window functions, it is possible to write out each one with a separate OVER clause, but this is duplicative and error-prone if the same windowing behavior is wanted for several functions. Instead, each windowing behavior can be named in a WINDOW clause and then referenced in OVER. For example:

```
SELECT sum(salary) OVER w, avg(salary) OVER w
  FROM empsalary
  WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);
```

More details about window functions can be found in ???, ???, and the ??? reference page.

Inheritance

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

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

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

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

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

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

A better solution is this:

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

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

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

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

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

which returns:

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

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

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


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

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

Note

Although inheritance is frequently useful, it has not been integrated with unique constraints or foreign keys, which limits its usefulness. See ??? for more detail.

Conclusion

PostgreSQL™ has many features not touched upon in this tutorial introduction, which has been oriented toward newer users of SQL. These features are discussed in more detail in the remainder of this book.

If you feel you need more introductory material, please visit the PostgreSQL web site [http://-www.postgresql.org] for links to more resources.

Bibliography

Selected references and readings for SQL and PostgreSQL™.

Some white papers and technical reports from the original POSTGRES™ development team are available at the University of California, Berkeley, Computer Science Department web site [http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/-papers/].

SQL Reference Books

Reference texts for SQL features.

[BOWMAN01] *The Practical SQL Handbook*. Bowman et al, 2001. Using SQL Variants. Fourth Edition. Judith Bowman, Sandra Emerson, and Marcy Darnovsky. 0-201-70309-2. 2001. Addison-Wesley Professional. Copyright © 2001.

[DATE97] *A Guide to the SQL Standard*. Date and Darwen, 1997. A user's guide to the standard database language SQL. Fourth Edition. C. J. Date and Hugh Darwen. 0-201-96426-0. 1997. Addison-Wesley. Copyright © 1997 Addison-Wesley Longman, Inc..

[DATE04] *An Introduction to Database Systems*. Date, 2004. Eighth Edition. C. J. Date. 0-321-19784-4. 2003. Addison-Wesley. Copyright © 2004 Pearson Education, Inc..

[ELMA04] *Fundamentals of Database Systems*. Fourth Edition. Ramez Elmasri and Shamkant Navathe. 0-321-12226-7. 2003. Addison-Wesley. Copyright © 2004.

[MELT93] *Understanding the New SQL*. Melton and Simon, 1993. A complete guide. Jim Melton and Alan R. Simon. 1-55860-245-3. 1993. Morgan Kaufmann. Copyright © 1993 Morgan Kaufmann Publishers, Inc..

[ULL88] *Principles of Database and Knowledge*. Base Systems. Ullman, 1988. Jeffrey D. Ullman. Volume 1. Computer Science Press. 1988.

PostgreSQL-Specific Documentation

This section is for related documentation.

[SIM98] *Enhancement of the ANSI SQL Implementation of PostgreSQL*. Simkovics, 1998. Stefan Simkovics. November 29, 1998. Department of Information Systems, Vienna University of Technology. Vienna, Austria.

[YU95] *The Postgres95. User Manual*. Yu and Chen, 1995. A. Yu and J. Chen. . Sept. 5, 1995. University of California. Berkeley, California.

[FONG] *The design and implementation of the POSTGRES™ query optimizer [http://s2k-ftp.CS.Berkeley.EDU:8000/-postgres/papers/UCB-MS-zfong.pdf]*. Zelaine Fong. University of California, Berkeley, Computer Science Department.

Proceedings and Articles

This section is for articles and newsletters.

[OLSON93] *Partial indexing in POSTGRES: research project*. Olson, 1993. Nels Olson. 1993. UCB Engin T7.49.1993 O676. University of California. Berkeley, California.

[ONG90] “A Unified Framework for Version Modeling Using Production Rules in a Database System”. Ong and Goh, 1990. L. Ong and J. Goh. *ERL Technical Memorandum M90/33*. April, 1990. University of California. Berkely, California.

[ROWE87] “ The POSTGRES™ data model [http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M87-13.pdf]”. Rowe and Stonebraker, 1987. L. Rowe and M. Stonebraker. VLDB Conference. Sept. 1987. Brighton, England. .

[SESHADRI95] “Generalized Partial Indexes (cached version) [http://citeseer.ist.psu.edu/seshadri95generalized.html] ”. Seshardri, 1995. P. Seshadri and A. Swami. Eleventh International Conference on Data Engineering. 6-10 March 1995. Taipeh, Taiwan. . 1995. Cat. No.95CH35724. IEEE Computer Society Press. Los Alamitos, California. 420-7.

[STON86] “ The design of POSTGRES™ [http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M85-95.pdf]”. Stonebraker and Rowe, 1986. M. Stonebraker and L. Rowe. ACM-SIGMOD Conference on Management of Data. May 1986. Washington, DC. .

[STON87a] “The design of the POSTGRES. rules system”. Stonebraker, Hanson, Hong, 1987. M. Stonebraker, E. Hanson, and C. H. Hong. IEEE Conference on Data Engineering. Feb. 1987. Los Angeles, California. .

[STON87b] “ The design of the POSTGRES™ storage system [http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/-ERL-M87-06.pdf]”. Stonebraker, 1987. M. Stonebraker. VLDB Conference. Sept. 1987. Brighton, England. .

[STON89] “ A commentary on the POSTGRES™ rules system [http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/-ERL-M89-82.pdf]”. Stonebraker et al, 1989. M. Stonebraker, M. Hearst, and S. Potamianos. *SIGMOD Record 18(3)*. Sept. 1989.

[STON89b] “ The case for partial indexes [http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-17.pdf]”. Stonebraker, M, 1989b. M. Stonebraker. *SIGMOD Record 18(4)*. 4-11. Dec. 1989.

[STON90a] “ The implementation of POSTGRES™ [http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/-ERL-M90-34.pdf]”. Stonebraker, Rowe, Hirohama, 1990. M. Stonebraker, L. A. Rowe, and M. Hirohama. *Transactions on Knowledge and Data Engineering 2(1)*. IEEE. March 1990.

[STON90b] “ On Rules, Procedures, Caching and Views in Database Systems [http://s2k-ftp.CS.Berkeley.EDU:8000/-postgres/papers/ERL-M90-36.pdf]”. Stonebraker et al, ACM, 1990. M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. ACM-SIGMOD Conference on Management of Data. June 1990. .