PostgreSQL Tutorial

The PostgreSQL Development Team

PostgreSQL Tutorial
Covering v6.3 for general release
by The PostgreSQL Development Team

Edited by Thomas Lockhart

(last updated 1998-02-23)

PostgreSQL is copyright (C) 1998 by the Postgres Global Development Group.

Table of Contents

Summary ..... *i*

1. Introduction 1

What is Postgres? 1
A Short History of Postgres 1
The Berkeley Postgres Project 1
Postgres95 2
PostgreSQL 2
About This Release 3
Resources 3
Copyrights and Trademarks 4

2. Architecture **5**

Postgres Architectural Concepts 5

3. Getting Started **6**

Setting Up Your Environment 6
Starting the Interactive Monitor (psql) 7
Managing a Database 7
Creating a Database 7
Accessing a Database 8
Destroying a Database 9

4. The Query Language **10**

Interactive Monitor 10
Concepts 10
Creating a New Class 11
Populating a Class with Instances 11
Querying a Class 11
Redirecting SELECT Queries 12
Joins Between Classes 12
Updates 13
Deletions 13
Using Aggregate Functions 13

5. Advanced Postgres SQL Features **15**

Inheritance 15
Non-Atomic Values 16
Arrays 16
Time Travel 17
More Advanced Features 18

SQL References **19**

List of Figures

2-1. How a connection is established 5

Summary

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

Chapter 1. Introduction

This document is the user manual for the PostgreSQL database management system, originally developed at the University of California at Berkeley. PostgreSQL is based on Postgres release 4.2. The Postgres project, led by Professor Michael Stonebraker, has been sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc.

What is Postgres?

Traditional relational database management systems (DBMSs) 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 four additional basic concepts in such a way that users can easily extend the system:

classes
inheritance
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.

A Short History of Postgres

The Berkeley Postgres Project

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

Postgres has undergone several major releases since then. The first "demoware" system became operational in 1987 and was shown at the 1988 ACM-SIGMOD Conference. We released Version 1, described in [STON90a], to a few external users in June 1989. In response

to a critique of the first rule system ([STON89]), the rule system was redesigned ([STON90b]) and Version 2 was released in June 1990 with the new rule system. Version 3 appeared in 1991 and added support for multiple storage managers, an improved query executor, and a rewritten rewrite rule system. For the most part, releases since then have 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 picked up the code and commercialized it. Postgres became the primary data manager for the Sequoia 2000 scientific computing project in late 1992. Furthermore, 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.

Postgres95

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

Postgres95 is a derivative of the last official release of Postgres (version 4.2). The code is now completely ANSI C and the code size has been trimmed by 25%. There are a lot of internal changes that improve performance and code maintainability. Postgres95 v1.0.x runs about 30-50% faster on the Wisconsin Benchmark compared to v4.2. Apart from bug fixes, these are the major enhancements:

The query language Postquel has been replaced with SQL (implemented in the server). We do not yet support subqueries (which can be imitated with user defined SQL functions). Aggregates have been re-implemented. We also added support for ‘‘GROUP BY’’. The libpq interface is still available for C programs.

In addition to the monitor program, we provide a new program (psql) which supports GNU readline.

We added a new front-end library, libpgtcl, that supports Tcl-based clients. A sample shell, pgtclsh, provides new Tcl commands to interface tcl programs with the Postgres95 backend.

The large object interface has been overhauled. We kept Inversion large objects as the only mechanism for storing large objects. (This is not to be confused with the Inversion file system which has been removed.)

The instance-level rule system has been removed. Rules are still available as rewrite rules.

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

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

PostgreSQL

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

The emphasis on development for the v1.0.x releases of Postgres95 was on stabilizing the backend code. With the v6.x series of PostgreSQL, the emphasis has shifted from identifying and understanding existing problems in the backend to augmenting features and capabilities, although work continues in all areas.

Major enhancements include:

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 coersion, 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%, and backend startup speed has decreased 80%.

About This Release

From now on, We will use Postgres to mean PostgreSQL.

PostgreSQL is available without cost. This manual describes version 6.3 of PostgreSQL.

Check the Administrator's Guide for a list of currently supported machines. In general, PostgreSQL is portable to any Unix/Posix-compatible system with full libc library support.

Resources

This manual set is organized into several parts:

Tutorial

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

User's Guide

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

Programmer's Guide

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

Administrator's Guide

Installation and management information. List of supported machines.

Developer's Guide

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

Reference Manual

Detailed reference information on command syntax. At the moment, this manual is very sparse, but eventually should contain information similar to that in the man pages.

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

man pages

The man pages have general information on command syntax.

FAQs

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

READMEs

README files are available for some contributed packages.

Web Site

The Postgres web site has some information not appearing in the distribution. There is a mhonarc catalog of mailing list traffic which is a rich resource for many topics.

Mailing Lists

The Postgres Questions mailing list is a good place to have user questions answered. Other mailing lists are available; consult the web page for details.

Yourself!

Postgres is an open source product. As such, it depends on the user community for ongoing support. As you begin to use Postgres, you will rely on others for help, either through the documentation or through the mailing lists. Consider contributing your knowledge back. If you learn something which is not in the documentation, write it up and contribute it. If you add features to the code, contribute it. Even those without a lot of experience can provide corrections and minor changes in the documentation, and that is a good way to start. The Postgres Documentation mailing list is a good place to get going.

Copyrights and Trademarks

PostgreSQL is copyright (C) 1996-8 by the PostgreSQL Global Development Group, and is distributed under the terms of the Berkeley license.

Postgres95 is copyright (C) 1994-5 by the Regents of the University of California. Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

In no event shall the University of California be liable to any party for direct, indirect, special, incidental, or consequential damages, including lost profits, arising out of the use of this software and its documentation, even if the University of California has been advised of the possibility of such damage.

The University of California specifically disclaims any warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The software provided hereunder is on an "as-is" basis, and the University of California has no obligations to provide maintainance, support, updates, enhancements, or modifications.

UNIX is a trademark of X/Open, Ltd. Sun4, SPARC, SunOS and Solaris are trademarks of Sun Microsystems, Inc. DEC, DECstation, Alpha AXP and ULTRIX are trademarks of Digital Equipment Corp. PA-RISC and HP-UX are trademarks of Hewlett-Packard Co. OSF/1 is a trademark of the Open Software Foundation.

Chapter 2. Architecture

Postgres Architectural Concepts

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

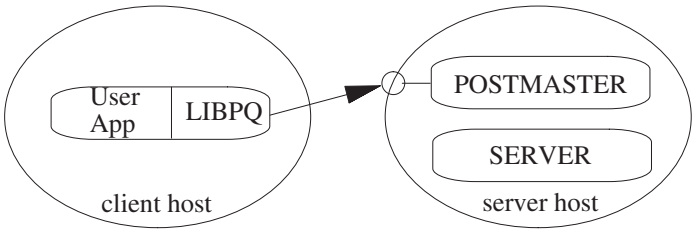
A supervisory daemon process (postmaster),

the user's frontend application (e.g., the psql program), and

the one or more backend database servers (the postgres process itself).

A single postmaster manages a given collection of databases on a single host. Such a collection of databases is called an installation or site. Frontend applications that wish to access a given database within an installation make calls to the library. The library sends user requests over the network to the postmaster (*How a connection is established*), which in turn starts a new backend server process

Figure 2-1. How a connection is established



and connects the frontend process to the new server. 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"). Furthermore, the Postgres superuser should definitely not be the UNIX superuser ("root")! In any case, all files relating to a database should belong to this Postgres superuser.

Chapter 3. Getting Started

How to begin work with Postgres for a new user.

Some of the steps required to use Postgres can be performed by any Postgres user, and some must be done by the site database administrator. This site administrator is the person who installed the software, created the database directories and started the postmaster process. This person does not have to be the UNIX superuser (root) or the computer system administrator; a person can install and use Postgres without any special accounts or privileges.

If you are installing Postgres yourself, then refer to the Administrator's Guide for instructions on installation, and return to this guide when the installation is complete.

Throughout this manual, any examples that begin with the character % are commands that should be typed at the UNIX shell prompt. Examples that begin with the character * are commands in the Postgres query language, Postgres SQL.

Setting Up Your Environment

This section discusses how to set up your own environment so that you can use frontend applications. We assume Postgres has already been successfully installed and started; refer to the Administrator's Guide and the installation notes for how to install Postgres.

Postgres is a client/server application. As a user, you only need access to the client portions of the installation (an example of a client application is the interactive monitor psql). For simplicity, we will assume that Postgres has been installed in the directory /usr/local/pgsql. Therefore, wherever you see the directory /usr/local/pgsql you should substitute the name of the directory where Postgres is actually installed. All Postgres commands are installed in the directory /usr/local/pgsql/bin. Therefore, you should add this directory to your shell command path. If you use a variant of the Berkeley C shell, such as csh or tcsh, you would add

```
% set path = ( /usr/local/pgsql/bin path )
```

in the .login file in your home directory. If you use a variant of the Bourne shell, such as sh, ksh, or bash, then you would add

```
% PATH=/usr/local/pgsql/bin PATH
% export PATH
```

to the .profile file in your home directory. From now on, we will assume that you have added the Postgres bin directory to your path. In addition, we will make frequent reference to setting a shell variable or setting an environment variable throughout this document. If you did not fully understand the last paragraph on modifying your search path, you should consult the UNIX manual pages that describe your shell before going any further.

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

Starting the Interactive Monitor (psql)

Assuming that your site administrator has properly started the postmaster process and authorized you to use the database, you (as a user) may begin to start up applications. As previously mentioned, you should add /usr/local/pgsql/bin to your shell search path. In most cases, this is all you should have to do in terms of preparation.

As of Postgres v6.3, two different styles of connections are supported. The site administrator will have chosen to allow TCP/IP network connections or will have restricted database access to local (same-machine) socket connections only. These choices become significant if you encounter problems in connecting to a database.

If you get the following error message from a Postgres command (such as psql or createdb):

```
% psql template1
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting connections
    at 'UNIX Socket' on port '5432'?
```

or

```
% psql -h localhost template1
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting TCP/IP
    (with -i) connections at 'localhost' on port '5432'?
```

it is usually because (1) the postmaster is not running, or (2) you are attempting to connect to the wrong server host. If you get the following error message:

```
FATAL 1:Feb 17 23:19:55:process userid (2360) != database owner (268)
```

it means that the site administrator started the postmaster as the wrong user. Tell him to restart it as the Postgres superuser.

Managing a Database

Now that Postgres is up and running we can create some databases to experiment with. Here, we describe the basic commands for managing a database.

Most Postgres applications assume that the database name, if not specified, is the same as the name on your computer account.

If your database administrator has set up your account without database creation privileges, then she should have told you what the name of your database is. If this is the case, then you can skip the sections on creating and destroying databases.

Creating a Database

Let's say you want to create a database named mydb. You can do this with the following command:

```
% createdb mydb
```

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

```
% createdb mydb
WARN:user "your username" is not allowed to create/destroy databases
createdb: database creation failed on mydb.
```

Postgres allows you to create any number of databases at a given site and you automatically become the database administrator of the database you just created. Database names must have an alphabetic first character and are limited to 32 characters in length. Not every user has authorization to become a database administrator. If Postgres 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.

Accessing a Database

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

running the Postgres terminal monitor programs (e.g. psql) which allows you to interactively enter, edit, and execute SQL commands.

writing a C program using the LIBPQ subroutine library. This allows you to submit SQL commands from C and get answers and status messages back to your program. This interface is discussed further in [xref to BIBLIOENTRY unsupported].

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

```
% psql mydb
```

You will be greeted with the following message:

```
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: template1

mydb=>
```

This prompt indicates that the terminal monitor is listening to you and that you can type SQL queries into a workspace maintained by the terminal monitor. The psql program responds to escape codes that begin with the backslash character, \ For example, you can get help on the syntax of various Postgres SQL commands by typing:

```
mydb=> \h
```

Once you have finished entering your queries into the workspace, you can pass the contents of the workspace to the Postgres server by typing:

```
mydb=> \g
```

This tells the server to process the query. If you terminate your query with a semicolon, the \g is not necessary. psql will automatically process semicolon terminated queries. To read queries from a file, say myFile, instead of entering them interactively, type:

```
mydb=> \i fileName
```

To get out of psql and return to UNIX, type

```
mydb=> \q
```

and psql will quit and return you to your command shell. (For more escape codes, type \h at the monitor prompt.) White space (i.e., spaces, tabs and newlines) may be used freely in SQL queries. Single-line comments are denoted by -- . Everything after the dashes up to the end of the line is ignored. Multiple-line comments, and comments within a line, are denoted by /* ... */

Destroying a Database

If you are the database administrator for the database mydb, you can destroy it using the following UNIX command:

```
% destroydb mydb
```

This action physically removes all of the UNIX files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

Chapter 4. The Query Language

The Postgres query language is a variant of the SQL3 draft next-generation standard. It has many extensions such as an extensible type system, inheritance, functions and production rules. These are features carried over from the original Postgres query language, PostQuel. This section provides an overview of how to use Postgres SQL to perform simple operations. This manual is only intended to give you an idea of our flavor of SQL 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 language features are not part of the ANSI standard.

Interactive Monitor

In the examples that follow, we assume that you have created the mydb database as described in the previous subsection and have started psql. Examples in this manual can also be found in /usr/local/pgsql/src/tutorial/. Refer to the README file in that directory for how to use them. To start the tutorial, do the following:

```
% cd /usr/local/pgsql/src/tutorial
% psql -s mydb
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: postgres

mydb=> \i basics.sql
```

The \i command read in queries from the specified files. The -s option puts you in single step mode which pauses before sending a query to the backend. Queries in this section are in the file basics.sql.

psql has a variety of \d commands for showing system information. Consult these commands for more details; for a listing, type \? at the psql prompt.

Concepts

The fundamental notion in Postgres is that of a class, which is a named collection of object instances. Each instance has the same collection of named attributes, and each attribute is of a specific type. Furthermore, each instance has a permanent object identifier (OID) that is unique throughout the installation. Because SQL syntax refers to tables, we will use the terms table and class interchangeably. Likewise, an SQL row is an instance and SQL columns are attributes. As previously discussed, classes are grouped into databases, and a collection of databases managed by a single postmaster process constitutes an installation or site.

Creating a New Class

You can create a new class by specifying the class name, along with all attribute 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
);
```

Note that both keywords and identifiers are case-insensitive; identifiers can become case-sensitive by surrounding them with double-quotes as allowed by SQL92. Postgres SQL supports the usual SQL types int, float, real, smallint, char(N), varchar(N), date, time, and timestamp, as well as other types of general utility and a rich set of geometric types. As we will see later, Postgres can be customized with an arbitrary number of user-defined data types. Consequently, type names are not syntactical keywords, except where required to support special cases in the SQL92 standard. So far, the Postgres create command looks exactly like the command used to create a table in a traditional relational system. However, we will presently see that classes have properties that are extensions of the relational model.

Populating a Class with Instances

The insert statement is used to populate a class with instances:

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

You can also use the copy command to perform load large amounts of data from flat (ASCII) files.

Querying a Class

The weather class can be queried with normal relational selection and projection queries. A SQL select statement is used to do this. The statement is divided into a target list (the part that lists the attributes to be returned) and a qualification (the part that specifies any restrictions). For example, to retrieve all the rows of weather, type:

```
SELECT * FROM WEATHER;
```

and the output should be:

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

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

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

Arbitrary Boolean operators (and, or and not) are allowed in the qualification of any query. For example,

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

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

As a final note, you can specify that the results of a select can be returned in a sorted order or with duplicate instances removed.

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

Redirecting SELECT Queries

Any select query can be redirected to a new class

```
SELECT * INTO TABLE temp FROM weather;
```

This forms an implicit create command, creating a new class temp with the attribute names and types specified in the target list of the select into command. We can then, of course, perform any operations on the resulting class that we can perform on other classes.

Joins Between Classes

Thus far, our queries have only accessed one class at a time. Queries can access multiple classes at once, or access the same class in such a way that multiple instances of the class are being processed at the same time. A query that accesses multiple instances of the same or different classes at one time is called a join query. As an example, say we wish to find all the records that are in the temperature range of other records. In effect, we need to compare the temp_lo and temp_hi attributes of each EMP instance to the temp_lo and temp_hi attributes of all other EMP instances.

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

We can do this with the following query:

```
SELECT W1.city, W1.temp_lo, W1.temp_hi,
    W2.city, W2.temp_lo, W2.temp_hi
    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   |
+--------------+-----+------+---------------+-----+------+
|San Francisco | 37  | 54   | San Francisco | 46  | 50   |
+--------------+-----+------+---------------+-----+------+
```

> NOTE: The semantics of such a join are that the qualification is a truth expression defined for the Cartesian product of the classes indicated in the query. For those instances in the Cartesian product for which the qualification is true, Postgres computes and returns the values specified in the target list. Postgres SQL does not assign any meaning to duplicate values in such expressions. This means that Postgres sometimes recomputes the same target list several times; this frequently happens when Boolean expressions are connected with an "or". To remove such duplicates, you must use the select distinct statement.

In this case, both W1 and W2 are surrogates for an instance of the class weather, and both range over all instances of the class. (In the terminology of most database systems, W1 and W2 are known as range variables.) A query can contain an arbitrary number of class names and surrogates.

Updates

You can update existing instances using the update command. Suppose you discover the temperature readings are all off by 2 degrees as of Nov 28, you may update the data as follow:

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

Deletions

Deletions are performed using the delete command:

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

All weather recording belongs to Hayward is removed. One should be wary of queries of the form

```
DELETE FROM classname;
```

Without a qualification, delete will simply remove all instances of the given class, leaving it empty. The system will not request confirmation before doing this.

Using Aggregate Functions

Like most other query languages, PostgreSQL supports aggregate functions. The current implementation of Postgres aggregate functions have some limitations. Specifically, while

there are aggregates to compute such functions as the count, sum, avg (average), max (maximum) and min (minimum) over a set of instances, aggregates can only appear in the target list of a query and not directly in the qualification (the where clause). As an example,

```
SELECT max(temp_lo) FROM weather;
```

is allowed, while

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

is not. However, as is often the case the query can be restated to accomplish the intended result; here by using a subselect:

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

Aggregates may also have group by clauses:

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

Chapter 5. Advanced Postgres SQL Features

Having covered the basics of using Postgres SQL to access your data, we will now discuss those features of Postgres that distinguish it from conventional data managers. These features include inheritance, time travel and non-atomic data values (array- and set-valued attributes). Examples in this section can also be found in advance.sql in the tutorial directory. (Refer to Chapter 4 for how to use it.)

Inheritance

Let's create two classes. The capitals class contains state capitals which are also cities. Naturally, the capitals class should inherit from cities.

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

CREATE TABLE capitals (
    state           char2
) INHERITS (cities);
```

In this case, an instance of capitals inherits all attributes (name, population, and altitude) from its parent, cities. The type of the attribute name is text, a native Postgres type for variable length ASCII strings. The type of the attribute population is float, a native Postgres type for double precision floating point numbers. State capitals have an extra attribute, state, that shows their state. In Postgres, a class can inherit from zero or more other classes, and a query can reference either all instances of a class or all instances of a class plus all of its descendants.

NOTE: The inheritance hierarchy is a directed acyclic graph.

For example, the following query finds all the cities that are situated at an attitude of 500ft or higher:

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

+----------+----------+
|name      | altitude |
+----------+----------+
|Las Vegas | 2174     |
+----------+----------+
|Mariposa  | 1953     |
+----------+----------+
```

On the other hand, to find the names of all cities, including state capitals, that are located at an altitude over 500ft, the query is:

```
SELECT c.name, c.altitude
    FROM cities* c
    WHERE c.altitude > 500;
```

which returns:

```
+----------+----------+
|name      | altitude |
+----------+----------+
|Las Vegas | 2174     |
+----------+----------+
|Mariposa  | 1953     |
+----------+----------+
|Madison   | 845      |
+----------+----------+
```

Here the * after cities indicates that the query should be run over cities and all classes below cities in the inheritance hierarchy. Many of the commands that we have already discussed (select, update and delete) support this * notation, as do others, like alter.

Non-Atomic Values

One of the tenets of the relational model is that the attributes of a relation are atomic. Postgres does not have this restriction; attributes can themselves contain sub-values that can be accessed from the query language. For example, you can create attributes that are arrays of base types.

Arrays

Postgres allows attributes of an instance to be defined as fixed-length or variable-length multi-dimensional arrays. Arrays of any base type or user-defined type can be created. To illustrate their use, we first create a class with arrays of base types.

```
CREATE TABLE SAL_EMP (
    name            text,
    pay_by_quarter  int4[],
    schedule        char16[][]
);
```

The above query will create a class named SAL_EMP with a text string (name), a one-dimensional array of int4 (pay_by_quarter), which represents the employee's salary by quarter and a two-dimensional array of char16 (schedule), which represents the employee's weekly schedule. Now we do some INSERTSs; note that when appending to an array, we enclose the values within braces and separate them by commas. If you know C, this is not unlike the syntax for initializing structures.

```
INSERT INTO SAL_EMP
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {}}');

INSERT INTO SAL_EMP
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"talk", "consult"}, {"meeting"}}');
```

By default, Postgres uses the "one-based" numbering convention for arrays -- that is, an array of n elements starts with array[1] and ends with array[n]. Now, we can run some queries on SAL_EMP. First, we show how to access a single element of an array at a time. This query retrieves the names of the employees whose pay changed in the second quarter:

```
SELECT name
    FROM SAL_EMP
    WHERE SAL_EMP.pay_by_quarter[1] <>
    SAL_EMP.pay_by_quarter[2];

+------+
|name  |
+------+
|Carol |
+------+
```

This query retrieves the third quarter pay of all employees:

```
SELECT SAL_EMP.pay_by_quarter[3] FROM SAL_EMP;


+---------------+
|pay_by_quarter |
+---------------+
|10000          |
+---------------+
|25000          |
+---------------+
```

We can also access arbitrary slices of an array, or subarrays. This query retrieves the first item on Bill's schedule for the first two days of the week.

```
SELECT SAL_EMP.schedule[1:2][1:1]
    FROM SAL_EMP
    WHERE SAL_EMP.name = 'Bill';

+--------------------+
|schedule            |
+--------------------+
|{{"meeting"},{""}} |
+--------------------+
```

Time Travel

As of Postgres v6.2, time travel is no longer supported. There are several reasons for this: performance impact, storage size, and a pg_time file which grows toward infinite size in a short period of time.

New features such as triggers allow one to mimic the behavior of time travel when desired, without incurring the overhead when it is not needed (for most users, this is most of the time). See examples in the contrib directory for more information.

> Time travel is deprecated: The remaining text in this section is retained only until it can be rewritten in the context of new techniques to accomplish the same purpose. Volunteers? - thomas 1998-01-12

Postgres supports the notion of time travel. This feature allows a user to run historical queries. For example, to find the current population of Mariposa city, one would query:

```
SELECT * FROM cities WHERE name = 'Mariposa';

+---------+------------+----------+
|name     | population | altitude |
+---------+------------+----------+
|Mariposa | 1320       | 1953     |
+---------+------------+----------+
```

Postgres will automatically find the version of Mariposa's record valid at the current time. One can also give a time range. For example to see the past and present populations of Mariposa, one would query:

```
SELECT name, population
    FROM cities['epoch', 'now']
    WHERE name = 'Mariposa';
```

where "epoch" indicates the beginning of the system clock. If you have executed all of the examples so far, then the above query returns:

```
+---------+------------+
|name     | population |
+---------+------------+
|Mariposa | 1200       |
+---------+------------+
|Mariposa | 1320       |
+---------+------------+
```

The default beginning of a time range is the earliest time representable by the system and the default end is the current time; thus, the above time range can be abbreviated as ''[,].''

More Advanced Features

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

Appendix

Additional related information.

SQL References

Selected references and readings for SQL and Postgres.

SQL Reference Books

Reference texts for SQL features.

The Practical SQL Handbook, Judity Bowman, Sandra Emerson, and Marcy Damovsky, ISBN: 0-201-44787-8, Addison-Wesley, 1997.

A Guide to The SQL Standard, C. J. Date and Hugh Darwen, ISBN: 0-201-96426-0, Addison-Wesley, 1997.

Understanding the New SQL, Jim Melton and Alan R. Simon, ISBN: 1-55860-245-3, Morgan Kaufmann, 1993.

PostgreSQL-Specific Documentation

This section is for related documentation.

The PostgreSQL Administrator's Guide, The PostgreSQL Global Development Group.

The PostgreSQL Programmer's Guide, The PostgreSQL Global Development Group.

The PostgreSQL Reference Manual, The PostgreSQL Global Development Group.

The PostgreSQL Tutorial Introduction, The PostgreSQL Global Development Group.

The PostgreSQL User's Guide, The PostgreSQL Global Development Group.

The Postgres95 User Manual, A. Yu and J. Chen, The POSTGRES Group, University of California, Berkeley CA.

Proceedings and Articles

This section is for articles and newsletters.

A Unified Framework for Version Modeling Using Production Rules in a Database System, L. Ong and J. Goh, ISSN: ERL Technical Memorandum M90/33, University of California, Berkeley CA.

The Postgres Data Model, L. Rowe and M. Stonebraker.

The Design of Postgres, M. Stonebraker and L. Rowe.

The Design of the Postgres Rules System, M. Stonebraker, E. Hanson, and C. H. Hong.

The Postgres Storage System, M. Stonebraker.

A Commentary on the Postgres Rules System, M. Stonebraker, M. Hearst, and S. Potamianos.

The Implementation of Postgres, M. Stonebraker, L. A. Rowe, and M. Hirohama.

On Rules, Procedures, Caching and Views in Database Systems, M. Stonebraker and et. al..