PostgreSQL User’s Guide

The PostgreSQL Development Team

PostgreSQL User's Guide
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. Setting Up Your Environment **5**

3. Managing a Database **6**

- Database Creation 6
- Alternate Database Locations 6
- Accessing a Database 7
  - Database Privileges 8
  - Table Privileges 8
- Destroying a Database 8

4. Data Types **9**

- Numeric Types 10
- Monetary Type 11
- Character Types 11
- Date/Time Types 11
  - Date/Time Styles 13
  - Time Zones 13
  - Date/Time Input 14
  - datetime 14
  - timespan 15
  - abstime 15
  - reltime 15
  - timestamp 16
  - interval 16
  - tinterval 16
- Boolean Type 16
- Geometric Types 17
  - Point 17
  - Line Segment 17
  - Box 18
  - Path 18
  - Polygon 18
  - Circle 19

5. Operators **20**

6. Functions **23**

7. Arrays **26**

8. Inheritance **28**

9. The Query Language **30**
Concepts 30
Creating a New Class 30
Populating a Class with Instances 31
Querying a Class 31
Redirecting SELECT Queries 32
Joins Between Classes 32
Updates 33
Deletions 33
Using Aggregate Functions 33
10. Disk Storage **34**
11. psql **35**
12. pgaccess **36**
SQL References **37**

List of Tables

4-1. Postgres Data Types 9
4-2. Postgres Function Constants 10
4-3. Postgres Numeric Types 10
4-4. Postgres Numeric Types 11
4-5. Postgres Character Types 11
4-6. Postgres Specialty Character Types 11
4-7. Postgres Date/Time Types 12
4-8. Postgres Date/Time Ranges 12
4-9. Postgres Date Styles 13
4-10. Postgres Date Order Conventions 13
4-11. Postgres Date/Time Special Constants 14
4-12. Postgres Boolean Type 16
4-13. Postgres Geometric Types 17
5-1. Postgres Operators 20
5-2. Postgres Numerical Operators 21
5-3. Postgres Time Interval Operators 21
5-4. Postgres Geometric Operators 22
6-1. Mathematical Functions 23
6-2. String Functions 23
6-3. Date/Time Functions 24
6-4. Geometric Functions 25
6-5. SQL92 Text Functions 25

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 the 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. 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.

Chapter 3. Managing a Database

NOTE: This section is currently a thinly disguised copy of the Tutorial. Needs to be augmented. - thomas 1998-01-12

Although the site administrator is responsible for overall management of the Postgres installation, some databases within the installation may be managed by another person, designated the database administrator. This assignment of responsibilities occurs when a database is created. A user may be assigned explicit privileges to create databases and/or to create new users. A user assigned both privileges can perform most administrative task within Postgres, but will not by default have the same operating system privileges as the site administrator.

The Database Administrator's Guide covers these topics in more detail.

Database Creation

Databases are created by the create database issued from within Postgres. createdb is a command-line utility provided to give the same functionality from outside Postgres.

The Postgres backend must be running for either method to succeed, and the user issuing the command must be the Postgres superuser or have been assigned database creation privileges by the superuser.

To create a new database named mydb from the command line, type

```
% createdb mydb
```

and to do the same from within psql type

```
* CREATE DATABASE mydb;
```

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

```
% 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.

Alternate Database Locations

It is possible to create a database in a location other than the default location for the installation. Remember that all database access actually occurs through the database backend, so that any location specified must be accessible by the backend.

Either an absolute path name or an environment variable may be specified as a location. Any environment variable specifying an alternate location must have been defined before the backend was started. Consult with the site administrator regarding preconfigured alternate database locations.

NOTE: The environment variable style of specification is to be preferred since it allows the site administrator more flexibility in managing disk storage.

For security and integrity reasons, any path or environment variable specified has some additional path fields appended.

Alternate database locations must be prepared by running initlocation.

To create a data storage area in /alt/postgres/data, ensure that /alt/postgres already exists. From the command line, type

```
% initlocation /alt/postgres/data
Creating Postgres database system directory /alt/postgres/data

Creating Postgres database system directory /alt/postgres/data/base
```

To do the same using an environment variable PGDATA2, type

```
% initlocation $PGDATA2
Creating Postgres database system directory /alt/postgres/data

Creating Postgres database system directory /alt/postgres/data/base
```

To create a database in the alternate storage area /alt/postgres/data from the command line, type

```
% createdb -D /alt/postgres/data mydb
```

or

```
% createdb -D PGDATA2 mydb
```

and to do the same from within psql type

```
* CREATE DATABASE mydb WITH LOCATION = 'PGDATA2';
```

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.
```

If the specified location does not exist or the database backend does not have permission to access it or to write to directories under it, you will see the following:

```
% createdb -D /alt/postgres/data mydb
ERROR:  Unable to create database directory
/alt/postgres/data/base/mydb
createdb: database creation failed on mydb.
```

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 section ??.

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 `/* ... */`

Database Privileges

Table Privileges

TBD

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. Data Types

Describes the built-in data types available in Postgres.

Table 4-1. Postgres Data Types

Postgres	SQL92 or SQL3	Description
bool	boolean	logical boolean (true/false)
box		rectangular box in 2D plane
char(n)	character(n)	fixed-length character string
circle		circle in 2D plane
date	date	calendar date without time of day
float4/8	float(p)	floating-point number with precision p
float8	real, double precision	double-precision floating-point number
int2	smallint	signed two-byte integer
int4	int, integer	signed 4-byte integer
int4	decimal(p,s)	exact numeric for p <= 9, s = 0
int4	numeric(p,s)	exact numeric for p == 9, s = 0
line		infinite line in 2D plane
lseg		line segment in 2D plane
money	decimal(9,2)	US-style currency
path		open and closed geometric path in 2D plane
point		geometric point in 2D plane
polygon		closed geometric path in 2D plane
time	time	time of day
timespan	interval	general-use time span
timestamp	timestamp with time zone	date/time
varchar(n)	character varying(n)	variable-length character string

Postgres has a rich set of native data types available to users. Users may add new types to Postgres using the define type command described elsewhere.

Table 4-2. Postgres Function Constants

Postgres	SQL92	Description
getpgusername()	current_user	user name in current session
date('now')	current_date	date of current transaction
time('now')	current_time	time of current transaction
timestamp('now')	current_timestamp	date and time of current transaction

Postgres has features at the forefront of ORDBMS development. In addition to SQL3 conformance, substantial portions of SQL92 are also supported. Although we strive for SQL92 compliance, there are some cases in the standard which are ill considered and which should not live through subsequent standards. Postgres will not make great efforts to conform to these cases. However, these cases tend to be little-used and obsure, and a typical user is not likely to run into them.

Although most of the input and output functions corresponding to the base types (e.g., integers and floating point numbers) do some error-checking, some are not particularly rigorous about it. More importantly, few of the operators and functions (e.g., addition and multiplication) perform any error-checking at all. Consequently, many of the numeric operators can (for example) silently underflow or overflow.

Some of the input and output functions are not invertible. That is, the result of an output function may lose precision when compared to the original input.

In the context of data types, the following sections will discuss SQL standards compliance, porting issues, and usage. Some Postgres types correspond directly to SQL92-compatible types. In other cases, data types defined by SQL92 syntax are mapped directly into native Postgres types. Many of the built-in types have obvious external formats. However, several types are either unique to Postgres, such as open and closed paths, or have several possibilities for formats, such as date and time types.

Numeric Types

Numeric types consist of two- and four-byte integers and four- and eight-byte floating point numbers.

Table 4-3. Postgres Numeric Types

Numeric Type	Storage	Description	Range
int2	2 bytes	Fixed-precision	-32768 to +32767
int4	4 bytes	Usual choice for fixed-precision	-2147483648 to +2147483647
float4	4 bytes	Variable-precision	7 decimal places
float8	8 bytes	Variable-precision	14 decimal places

The exact numerics decimal and numeric have fully implemented syntax but currently (Postgres v6.3) support only a small range of precision and/or range values.

Monetary Type

The money type supports US-style currency with fixed decimal point representation.

Table 4-4. Postgres Numeric Types

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

The exact numerics decimal and numeric have fully implemented syntax but currently (Postgres v6.3) support only a small range of precision and/or range values.

Character Types

SQL92 defines two primary character types: char and varchar. Postgres supports these types, in addition to the more general text type, which unlike varchar does not require an upper limit to be declared on the size of the field.

Table 4-5. Postgres Character Types

Character Type	Storage	Recommendation	Description
char	1 byte	SQL92-compatible	Single character
char(n)	(4+n) bytes	SQL92-compatible	Fixed-length blank padded
text	(4+x) bytes	Best choice	Variable-length
varchar(n)	(4+n) bytes	SQL92-compatible	Variable-length with limit

There are currently other fixed-length character types. These provide no additional functionality and are likely to be deprecated in the future.

Table 4-6. Postgres Specialty Character Types

Character Type	Storage	Description
char2	2 bytes	Two characters
char4	4 bytes	Four characters
char8	8 bytes	Eight characters
char16	16 bytes	Sixteen characters

Date/Time Types

There are two fundamental kinds of date and time measurements: clock time and time interval. Both quantities have continuity and smoothness, as does time itself. Postgres supplies two

primary user-oriented date and time types, datetime and timespan, as well as the related SQL92 types date and time.

Other date and time types are available also, mostly for historical reasons.

Table 4-7. Postgres Date/Time Types

Date/Time Type	Storage	Recommendation	Description
abstime	4 bytes	original date and time	limited range
date	4 bytes	SQL92 type	wide range
datetime	8 bytes	best general date and time	wide range, high precision
interval	12 bytes	SQL92 type	equivalent to timespan
reltime	4 bytes	original time interval	limited range, low precision
time	4 bytes	SQL92 type	wide range
timespan	12 bytes	best general time interval	wide range, high precision
timestamp	4 bytes	SQL92 type	limited range

Table 4-8. Postgres Date/Time Ranges

Date/Time Type	Earliest	Latest	Resolution
abstime	1901-12-14	2038-01-19	1 sec
date	4713 BC	no limit	1 day
datetime	4713 BC	no limit	1 microsec to 14 digits
interval	no limit	no limit	1 microsec
reltime	-68 years	+68 years	1 sec
time	00:00:00.00	23:59:59.99	1 microsec
timespan	no limit	no limit	1 microsec (14 digits)
timestamp	1901-12-14	2038-01-19	1 sec

Postgres endevours to be compatible with SQL92 definitions for typical usage. The SQL92 standard has an odd mix of date and time types and capabilities. For example, although the date type does not have an associated time zone, the time type can. The default time zone is specified as a constant offset from GMT/UTC; however, time zones in the real world can have no meaning unless associated with a date as well as a time since the offset will vary through the year.

To obviate these difficulties, Postgres associates time zones only with date and time types which contain both date and time, and assumes local time for any type containing only date or time. Further, time zone support is derived from the underlying operating system time zone capabilities, and hence can handle daylight savings time and other expected behavior.

In future releases, the number of date/time types will decrease, with the current implementation of datetime becoming timestamp, timespan becoming interval, and (possibly) abstime and reltime being deprecated in favor of timestamp and interval. The more arcane features of the date/time definitions from the SQL92 standard are not likely to be pursued.

Date/Time Styles

Output formats can be set to one of four styles: ISO-8601, SQL (Ingres), traditional Postgres, and German.

Table 4-9. Postgres Date Styles

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

The SQL style has European and non-European (US) variants, which determines whether month follows day or vica versa.

Table 4-10. Postgres Date Order Conventions

Style Specification	Description	Example
European	Regional convention	17/12/1997 15:37:16.00 MET
NonEuropean	Regional convention	12/17/1997 07:37:16.00 PST
US	Regional convention	12/17/1997 07:37:16.00 PST

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

The PGDATESTYLE environment variable used by the backend directly on postmaster startup.
The PGDATESTYLE environment variable used by the frontend libpq on session startup.
SET DateStyle SQL command.

For Postgres v6.3 (and earlier) the default date/time style is "traditional Postgres". In future releases, the default may become ISO-8601, which alleviates date specification ambiguities and Y2K collation problems.

Time Zones

Postgres obtains time zone support from the underlying operating system. All dates and times are stored internally in Universal Coordinated Time (UTC), alternately known as Greenwich Mean Time (GMT). Times are converted to local time on the database server before being sent to the client frontend, hence by default are in the server time zone.

There are several ways to affect the time zone behavior:

The TZ environment variable used by the backend directly on postmaster startup as the default time zone.
The PGTZ environment variable set at the client used by libpq to send time zone information to the backend upon connection.
set timezone SQL sets the time zone for the session.

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

Date/Time Input

General-use date and time is input using a wide range of styles, including ISO-compatible, SQL-compatible, traditional Postgres and other permutations of date and time. In cases where interpretation can be ambiguous (quite possible with many traditional styles of date specification) Postgres uses a style setting to resolve the ambiguity.

Most date and time types share code for data input. For those types the input can have any of a wide variety of styles. For numeric date representations, European and US conventions can differ, and the proper interpretation is obtained by using the set datestyle command before entering data. Note that the style setting does not preclude use of various styles for input; it is used primarily to determine the output style and to resolve ambiguities.

The special values ‘current’, ‘infinity’ and ‘-infinity’ are provided. ‘infinity’ specifies a time later than any other valid time, and ‘-infinity’ specifies a time earlier than any other valid time. ‘current’ indicates that the current time should be substituted whenever this value appears in a computation. The strings ‘now’, ‘today’, ‘yesterday’, ‘tomorrow’, and ‘epoch’ can be used to specify time values. ‘now’ means the current transaction time, and differs from ‘current’ in that the current time is immediately substituted for it. ‘epoch’ means Jan 1 00:00:00 1970 GMT.

Table 4-11. Postgres Date/Time Special Constants

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

datetime

General-use date and time is input using a wide range of styles, including ISO-compatible, SQL-compatible, traditional Postgres (see section on "absolute time") and other permutations of date and time. Output styles can be ISO-compatible, SQL-compatible, or traditional Postgres, with the default set to be compatible with Postgres v6.0.

datetime is specified using the following syntax:

```
Year-Month-Day [ Hour : Minute : Second ]      [AD,BC] [ Timezone ]
  YearMonthDay [ Hour : Minute : Second ]      [AD,BC] [ Timezone ]
     Month Day [ Hour : Minute : Second ] Year [AD,BC] [ Timezone ]
where
    Year is 4013 BC, ..., very large
    Month is Jan, Feb, ..., Dec or 1, 2, ..., 12
    Day is 1, 2, ..., 31
    Hour is 00, 02, ..., 23
    Minute is 00, 01, ..., 59
```

```
    Second is 00, 01, ..., 59 (60 for leap second)
    Timezone is 3 characters or ISO offset to GMT
```

Valid dates are from Nov 13 00:00:00 4013 BC GMT to far into the future. Timezones are either three characters (e.g. "GMT" or "PST") or ISO-compatible offsets to GMT (e.g. "-08" or "-08:00" when in Pacific Standard Time). Dates are stored internally in Greenwich Mean Time. Input and output routines translate time to the local time zone of the server.

timespan

General-use time span is input using a wide range of syntaxes, including ISO-compatible, SQL-compatible, traditional Postgres (see section on "relative time") and other permutations of time span. Output formats can be ISO-compatible, SQL-compatible, or traditional Postgres, with the default set to be Postgres-compatible. Months and years are a "qualitative" time interval, and are stored separately from the other "quantitative" time intervals such as day or hour. For date arithmetic, the qualitative time units are instantiated in the context of the relevant date or time.

Time span is specified with the following syntax:

```
  Quantity Unit [Quantity Unit...] [Direction]
@ Quantity Unit [Direction]
where
    Quantity is ..., ‘-1’, ‘0’, ‘1’, ‘2’, ...
    Unit is ‘second’, ‘minute’, ‘hour’, ’day’, ‘week’, ‘month’, ‘year’,
    ’decade’, ’century’, millenium’, or abbreviations or plurals of
these units.
    Direction is ‘ago’.
```

abstime

Absolute time (abstime) is a limited-range (+/- 68 years) and limited-precision (1 sec) date data type. datetime may be preferred, since it covers a larger range with greater precision.

Absolute time is specified using the following syntax:

```
Month  Day [ Hour : Minute : Second ]  Year [ Timezone ]
where
    Month is Jan, Feb, ..., Dec
    Day is 1, 2, ..., 31
    Hour is 01, 02, ..., 24
    Minute is 00, 01, ..., 59
    Second is 00, 01, ..., 59
    Year is 1901, 1902, ..., 2038
```

Valid dates are from Dec 13 20:45:53 1901 GMT to Jan 19 03:14:04 2038 GMT. As of Version 3.0, times are no longer read and written using Greenwich Mean Time; the input and output routines default to the local time zone. All special values allowed for datetime are also allowed for "absolute time".

reltime

Relative time reltime is a limited-range (+/- 68 years) and limited-precision (1 sec) time span data type. timespan should be preferred, since it covers a larger range with greater precision and, more importantly, can distinguish between relative units (months and years) and quantitative units (days, hours, etc). Instead, reltime must force months to be exactly 30 days, so time arithmetic does not always work as expected. For example, adding one reltime year to abstime today does not produce today’s date one year from now, but rather a date 360 days from today.

reltime shares input and output routines with the other time span types. The section on timespan covers this in more detail.

timestamp

This is currently a limited-range absolute time which closely resembles the abstime data type. It shares the general input parser with the other date/time types. In future releases this type will absorb the capabilities of the datetime type and will move toward SQL92 compliance.

timestamp is specified using the same syntax as for datetime.

interval

interval is an SQL92 data type which is currently mapped to the timespan Postgres data type.

tinterval

Time ranges are specified as:

```
[ 'abstime' 'abstime']
where
    abstime is a time in the absolute time format.
```

Special abstime values such as ‘current’, ‘infinity’ and ‘-infinity’ can be used.

Boolean Type

Postgres supports bool as the SQL3 boolean type. bool can have one of only two states: 'true' or 'false'. A third state, 'unknown', is not implemented and is not suggested in SQL3; NULL is an effective substitute. bool can be used in any boolean expression, and boolean expressions always evaluate to a result compatible with this type.

bool uses 4 bytes of storage.

Table 4-12. Postgres Boolean Type

State	Output	Input
True	't'	TRUE, 't', 'true', 'y', 'yes', '1'
False	'f'	FALSE, 'f', 'false', 'n', 'no', '0'

Geometric Types

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

Table 4-13. Postgres Geometric Types

Geometric Type	Storage	Representation	Description
point	16 bytes	(x,y)	Point in space
line	32 bytes	((x1,y1),(x2,y2))	Infinite line
lseg	32 bytes	((x1,y1),(x2,y2))	Finite line segment
box	32 bytes	((x1,y1),(x2,y2))	Rectangular box
path	4+32n bytes	((x1,y1),...)	Closed path (similar to polygon)
path	4+32n bytes	[(x1,y1),...]	Open path
polygon	4+32n bytes	((x1,y1),...)	Polygon (similar to closed path)
circle	24 bytes	<(x,y),r>	Circle (center and radius)

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

Point

Points are specified using the following syntax:

```
( x , y )
  x , y
where
    x is the x-axis coordinate as a floating point number
    y is the y-axis coordinate as a floating point number
```

Line Segment

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

lseg is specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
where
    (x1,y1) and (x2,y2) are the endpoints of the segment
```

Box

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

box is specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
where
    (x1,y1) and (x2,y2) are opposite corners
```

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

Path

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

path is specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
where
    (x1,y1),...,(xn,yn) are points 1 through n
    a leading "[" indicates an open path
    a leading "(" indicates a closed path
```

Paths are output using the first syntax. Note that Postgres versions prior to v6.1 used a format for paths which had a single leading parenthesis, a "closed" flag, an integer count of the number of points, then the list of points followed by a closing parenthesis. The built-in function upgradepath is supplied to convert paths dumped and reloaded from pre-v6.1 databases.

Polygon

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

polygon is specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
where
    (x1,y1),...,(xn,yn) are points 1 through n
```

Polygons are output using the first syntax. Note that Postgres versions prior to v6.1 used a format for polygons which had a single leading parenthesis, the list of x-axis coordinates, the list of y-axis coordinates, followed by a closing parenthesis. The built-in function upgradepoly is supplied to convert polygons dumped and reloaded from pre-v6.1 databases.

Circle

Circles are represented by a center point and a radius.

circle is specified using the following syntax:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
where
    (x,y) is the center of the circle
    r is the radius of the circle
```

Circles are output using the first syntax.

Chapter 5. Operators

Postgres provides a large number of built-in operators on system types. These operators are declared in the system catalog pg_operator. Every entry in pg_operator includes the name of the procedure that implements the operator and the class OIDs of the input and output types.

To view all variations of the || string concatenation operator, try

```
    SELECT oprleft, oprright, oprresult, oprcode
    FROM pg_operator WHERE oprname = '||';

oprleft|oprright|oprresult|oprcode
-------+--------+---------+-------
     25|      25|       25|textcat
   1042|    1042|     1042|textcat
   1043|    1043|     1043|textcat
(3 rows)
```

Table 5-1. Postgres Operators

Operator	Description	Usage
<	Less than?	1 < 2
<=	Less than or equal to?	1 <= 2
<>	Not equal?	1 <> 2
=	Equal?	1 = 1
>	Greater than?	2 > 1
>=	Greater than or equal to?	2 >= 1
||	Concatenate strings	'Postgre' || 'SQL'
!!=	NOT IN	3 !!= i
~~	LIKE	'scrappy,hermit' ~~ '%scrappy%'
!~~	NOT LIKE	'bruce' !~~ '%al%'
~	Match (regex), case sensitive	'thomas' ~ '*.thomas*.'
~*	Match (regex), case insensitive	'thomas' ~* '*.Thomas*.'
!~	Does not match, case sensitive	'thomas' !~ '*.Thomas*.'
!~*	Does not match, case insensitive	'thomas' !~ '*.vadim*.'

Table 5-2. Postgres Numerical Operators

Operator	Description	Usage
!	Factorial	3 !
!!	Factorial (left operator)	!! 3
%	Modulo	5 % 4
%	Truncate	% 4.5
*	Multiplication	2 * 3
+	Addition	2 + 3
-	Subtraction	2 - 3
/	Division	4 / 2
:	Natural Exponentiation	: 3.0
;	Natural Logarithm	(; 5.0)
@	Absolute value	@ -5.0
^	Exponentiation	2.0 ^ 3.0
|/	Square root	|/ 25.0
||/	Cube root	||/ 27.0

The time interval data type tinterval is a legacy from the original date/time types and is not as well supported as the more modern types. There are several operators for this type.

Table 5-3. Postgres Time Interval Operators

Operator	Description	Usage
#<	Interval less than?	
#<=	Interval less than or equal to?	
#<>	Interval not equal?	
#=	Interval equal?	
#>	Interval greater than?	
#>=	Interval greater than or equal to?	
<#>	Convert to time interval	
<<	Interval less than?	
|	Start of interval	
~=	Same as	
<?>	Time inside interval?	

Table 5-4. Postgres Geometric Operators

Operator	Description	Usage
+	Translation	’((0,0),(1,1))’::box + ’(2.0,0)’::point
-	Translation	’((0,0),(1,1))’::box - ’(2.0,0)’::point
*	Scaling/rotation	’((0,0),(1,1))’::box * ’(2.0,0)’::point
/	Scaling/rotation	’((0,0),(2,2))’::box / ’(2.0,0)’::point
#	Intersection	’((1,-1),(-1,1))’ # ’((1,1),(-1,-1))’
#	Number of points	# ’((1,0),(0,1),(-1,0))’
##	Point of closest proximity	’(0,0)’::point ## ’((2,0),(0,2))’::lseg
&&	Overlaps?	’((0,0),(1,1))’::box && ’((0,0),(2,2))’::box
&<	Overlaps to left?	’((0,0),(1,1))’::box &< ’((0,0),(2,2))’::box
&>	Overlaps to right?	’((0,0),(3,3))’::box &> ’((0,0),(2,2))’::box
<->	Distance between	’((0,0),1)’::circle <-> ’((5,0),1)’::circle
<<	Left of?	’((0,0),1)’::circle << ’((5,0),1)’::circle
<^	Is below?	’((0,0),1)’::circle <^ ’((0,5),1)’::circle
>>	Is right of?	’((5,0),1)’::circle >> ’((0,0),1)’::circle
>^	Is above?	’((0,5),1)’::circle >^ ’((0,0),1)’::circle
?#	Intersects or overlaps	’((-1,0),(1,0))’::lseg ?# ’((-2,-2),(2,2))’::box;
?-	Is horizontal?	’(1,0)’::point ?- ’(0,0)’::point
?-|	Is perpendicular?	’((0,0),(0,1))’::lseg ?-| ’((0,0),(1,0))’::lseg
@-@	Length or circumference	@-@ ’((0,0),(1,0))’::path
?|	Is vertical?	’(0,1)’::point ?| ’(0,0)’::point
?||	Is parallel?	’((-1,0),(1,0))’::lseg ?|| ’((-1,2),(1,2))’::lseg
@	Contained or on	’(1,1)’::point @ ’((0,0),2)’::circle
@@	Center of	@@ ’((0,0),10)’::circle
~=	Same as	’((0,0),(1,1))’::polygon ~= ’((1,1),(0,0))’::polygon

Users may invoke operators using the operator name, as in:

```
select * from emp where salary < 40000;
```

Alternatively, users may call the functions that implement the operators directly. In this case, the query above would be expressed as:

```
select * from emp where int4lt(salary, 40000);
```

psql has a \dd command to show these operators.

Chapter 6. Functions

Many data types have functions available for conversion to other related types. In addition, there are some type-specific functions. Functions which are also available through operators are documented as operators only.

Some functions defined for text are also available for char() and varchar().

For the date_part and date_trunc functions, arguments can be ‘year’, ‘month’, ‘day’, ‘hour’, ‘minute’, and ‘second’, as well as the more specialized quantities ‘decade’, ‘century’, ‘millenium’, ‘millisecond’, and ‘microsecond’. date_part allows ‘dow’ to return day of week and ‘epoch’ to return seconds since 1970 (for datetime) or ’epoch’ to return total elapsed seconds (for timespan).

Table 6-1. Mathematical Functions

Function	Returns	Description	Example
float(int)	float8	convert integer to floating point	float(2)
float4(int)	float4	convert integer to floating point	float4(2)
int	integer(float)	convert floating point to integer	integer(2.0)

Many of the string functions listed below are available for text, varchar(), and char() types. At the moment, some functions are available only for the text type.

Table 6-2. String Functions

Function	Returns	Description	Example
lower(text)	text	convert text to lower case	lower(’TOM’)
lpad(text,int,text)	text	left pad string to length	lpad(’hi’,4,’??’)
ltrim(text,text)	text	left trim characters from text	ltrim(’xxxxtrim’,’x’)
position(text,text)	text	extract specified substring	position(’high’,’ig’)
rpad(text,int,text)	text	right pad string to length	rpad(’hi’,4,’x’)
rtrim(text,text)	text	right trim characters	rtrim(’trimxxxx’,’x’)
substr(text,int[,int])	text	extract specified substring	substr(’hi there’,3,5)
upper(text)	text	convert text to upper case	upper(’tom’)

Table 6-3. Date/Time Functions

Function	Returns	Description	Example
isfinite(abstime)	bool	TRUE if this is a finite time	isfinite('now'::abstime)
datetime(abstime)	datetime	convert to datetime	datetime('now'::abstime)
datetime(date)	datetime	convert to datetime	datetime('today'::date)
datetime(date,time)	datetime	convert to datetime	datetime('1998-02-24':datetime, '23:07'::time);
age(datetime,datetime)	timespan	span preserving months and years	age('now','1957-06-13':datetime)
date_part(text,datetime)	float8	specified portion of date field	date_part('dow','now'::datetime)
date_trunc(text,datetime)	datetime	truncate date at specified units	date_trunc('month','now'::abstime)
isfinite(datetime)	bool	TRUE if this is a finite time	isfinite('now'::datetime)
abstime(datetime)	abstime	convert to abstime	abstime('now'::datetime)
timespan(reltime)	timespan	convert to timespan	timespan('4 hours'::reltime)
datetime(date,time)	datetime	convert to datetime	datetime('1998-02-25'::date, '06:41'::time)
date_part(text,timespan)	float8	specified portion of time field	date_part('hour', '4 hrs 3 mins'::timespan)
isfinite(timespan)	bool	TRUE if this is a finite time	isfinite('4 hrs'::timespan)
reltime(timespan)	reltime	convert to reltime	reltime('4 hrs'::timespan)

Table 6-4. Geometric Functions

Function	Returns	Description	Example
box(point,point)	box	convert points to box	box('(0,0)'::point,'(1,1)'::point)
area(box)	float8	area of box	area('((0,0),(1,1))'::box)
isopen(path)	bool	TRUE if this is an open path	isopen('[(0,0),(1,1),(2,0)]'::path)
isclosed(path)	bool	TRUE if this is a closed path	isclosed('((0,0),(1,1),(2,0))'::path)
circle(point,float8)	circle	convert to circle	circle('(0,0)'::point,2.0)
polygon(npts,circle)	polygon	convert to polygon with npts points	polygon(12,'((0,0),2.0)'::circle)
center(circle)	float8	center of object	center('((0,0),2.0)'::circle)
radius(circle)	float8	radius of circle	radius('((0,0),2.0)'::circle)
diameter(circle)	float8	diameter of circle	diameter('((0,0),2.0)'::circle)
area(circle)	float8	area of circle	area('((0,0),2.0)'::circle)

SQL92 defines functions with specific syntax. Some of these are implemented using other Postgres functions.

Table 6-5. SQL92 Text Functions

Function	Returns	Description	Example
position(text in text)	int4	extract substring	position('o' in 'Tom')
substring(text [from int] [for int])	text	extract substring	substring('Tom' from 2 for 2)
trim([leading|trailing|both] [text] from text)	text	trim characters	trim(both 'x' from 'xTomx')

Chapter 7. Arrays

NOTE: This must become a chapter on array behavior. Volunteers? - thomas 1998-01-12

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

Chapter 8. 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 actually 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.

Chapter 9. The Query Language

NOTE: This chapter must go into depth on each area of the query language. Currently a copy of the tutorial. - thomas 1998-01-12

The Postgres query language is a variant of SQL3. It has many extensions such as an extensible type system, inheritance, functions and production rules. Those 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. For instance, consult [MELT93] or [DATE93]. You should also be aware that some features are not part of the ANSI standard.

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 keywords are case-insensitive and identifiers are usually case-insensitive. Postgres allows SQL92 delimited identifiers (identifiers surrounded by double-quotes) to include mixed-case and spaces, tabs, etc.

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          | temp_lo | temp_hi | city          | temp_lo | temp_hi
|
+--------------+---------+---------+---------------+---------+---------
-+
|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 10. Disk Storage

This section needs to be written. Some information is in the FAQ. Volunteers? - thomas 1998-01-11

Chapter 11. psql

This section needs to be written. Volunteers?

Chapter 12. pgaccess

This section needs to be written. Volunteers?

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..