PostgreSQL User’s Guide

The PostgreSQL Development Team

Edited by
Thomas Lockhart

PostgreSQL User's Guide
by The PostgreSQL Development Team

Edited by Thomas Lockhart

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
Terminology 4
Notation 5
Y2K Statement 6
Copyrights and Trademarks 6

2. Setting Up Your Environment **7**

3. Managing a Database **8**

Database Creation 8
Alternate Database Locations 8
Accessing a Database 9
Database Privileges 10
Table Privileges 10
Destroying a Database 10

4. SQL Syntax **11**

Key Words 11
Reserved Key Words 11
Non-reserved Keywords 13

5. Data Types **15**

Numeric Types 17
Monetary Type 18
Character Types 18
Date/Time Types 19
Date/Time Styles 20
Time Zones 21
Date/Time Input 21
datetime 22
timespan 22
abstime 23
reltime 23
timestamp 23
interval 23
tinterval 23
Boolean Type 23
Geometric Types 24
Point 24
Line Segment 25
Box 25
Path 25
Polygon 26
Circle 26
IP Version 4 Networks and Host Addresses 26

inet for IP Networks 26
inet for IP Networks 27
6. Operators 28
Lexical Precedence 28
General Operators 30
Numerical Operators 31
Geometric Operators 31
Time Interval Operators 33
IP V4 Operators 34
7. Functions 35
Mathematical Functions 35
String Functions 36
Date/Time Functions 37
Geometric Functions 38
IP V4 Functions 40
8. Type Conversion 41
Overview 41
Guidelines 42
Operators 43
Conversion Procedure 43
Examples 43
Exponentiation Operator 43
String Concatenation 44
Factorial 44
Functions 45
Examples 45
Factorial Function 45
Substring Function 45
Query Targets 46
Examples 47
varchar Storage 47
UNION Queries 47
Examples 47
Underspecified Types 47
Simple UNION 47
Transposed UNION 47
9. Indices and Keys 49
10. Arrays 51
11. Inheritance 53
12. The Query Language 54
Concepts 54
Creating a New Class 54
Populating a Class with Instances 55
Querying a Class 55
Redirecting SELECT Queries 56
Joins Between Classes 56
Updates 57
Deletions 57
Using Aggregate Functions 57
13. Disk Storage 58
14. psql 59
Paging To Screen 59

15. pgaccess 60
16. SQL Commands 61
ABORT 61
ALTER TABLE 62
ALTER USER 65
BEGIN WORK 67
CLOSE 68
CLUSTER 69
COMMIT 71
COPY 72
CREATE AGGREGATE 76
CREATE DATABASE 79
CREATE FUNCTION 81
CREATE INDEX 83
CREATE LANGUAGE 85
CREATE OPERATOR 88
CREATE RULE 92
CREATE SEQUENCE 95
CREATE TABLE 99
CREATE TABLE AS 112
CREATE TRIGGER 113
CREATE TYPE 115
CREATE USER 119
CREATE VIEW 122
DECLARE 124
DELETE 127
DROP AGGREGATE 129
DROP DATABASE 130
DROP FUNCTION 131
DROP INDEX 133
DROP LANGUAGE 134
DROP OPERATOR 135
DROP RULE 137
DROP SEQUENCE 138
DROP TABLE 139
DROP TRIGGER 141
DROP TYPE 142
DROP USER 143
DROP VIEW 144
EXPLAIN 146
FETCH 147
GRANT 150
INSERT 154
LISTEN 156
LOAD 158
LOCK 160
MOVE 162
NOTIFY 163
RESET 165
REVOKE 166
ROLLBACK 169
SELECT 170
SELECT INTO 176
SET 177
SHOW 182

UNLISTEN 183
UPDATE 185
VACUUM 187
17. Utility Applications 189
createdb 189
createuser 191
destroydb 193
destroyuser 195
initdb 197
initlocation 200
pg_dump 202
pg_dumpall 205
psql 207
SQL References 214

List of Tables

5-1. Postgres Data Types 15
5-2. Postgres Function Constants 16
5-3. Postgres Numeric Types 17
5-4. Postgres Numeric Types 18
5-5. Postgres Character Types 18
5-6. Postgres Specialty Character Type 18
5-7. Postgres Date/Time Types 19
5-8. Postgres Date/Time Ranges 19
5-9. Postgres Date Styles 20
5-10. Postgres Date Order Conventions 20
5-11. Postgres Date/Time Special Constants 21
5-12. Postgres Boolean Type 24
5-13. Postgres Geometric Types 24
5-14. PostgresIP Version 4 Types 26
5-15. PostgresIP Types Examples 27
6-1. Operator Ordering (decreasing precedence) 29
6-2. Postgres Operators 30
6-3. Postgres Numerical Operators 31
6-4. Postgres Geometric Operators 31
6-5. Postgres Time Interval Operators 33
6-6. PostgresIP V4 Operators 34
7-1. Mathematical Functions 35
7-2. SQL92 String Functions 35
7-3. String Functions 36
7-4. Date/Time Functions 37
7-5. Geometric Functions 38
7-6. Geometric Type Conversion Functions 39
7-7. Geometric Upgrade Functions 40
7-8. PostgresIP V4 Functions 40
16-1. Contents of a binary copy file 74

List of Examples

16-1. Example of a circular rewrite rule combination. 93

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

Postgres has undergone several major releases since then. The first "demoware" system became operational in 1987 and was shown at the 1988 ACM-SIGMOD Conference. We released Version 1, described in *The Implementation of Postgres*, to a few external users in June 1989. In response to a critique of the first rule system (*A Commentary on the Postgres Rules System*),

the rule system was redesigned (*On Rules, Procedures, Caching and Views in Database Systems*) and Version 2 was released in June 1990 with the new rule system. Version 3 appeared in 1991 and added support for multiple storage managers, an improved query executor, and a rewritten rewrite rule system. For the most part, releases 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 (since merged into Informix) 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 20-40%, and backend startup time has decreased 80% since v6.0 was released.

About This Release

PostgreSQL is available without cost. This manual describes version 6.4 of PostgreSQL.

We will use Postgres to mean the version distributed as PostgreSQL.

Check the Administrator's Guide for a list of currently supported machines. In general, Postgres 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. Currently included in the Programmer's Guide.

Reference Manual

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

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

man pages

The man pages have general information on command syntax.

FAQs

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

READMEs

README files are available for some contributed packages.

Web Site

The Postgres 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.

Terminology

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

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

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

The postmaster is the process that acts as a clearing-house for requests to the Postgres system. Frontend applications connect to the postmaster, which keeps tracks of any system errors and communication between the backend processes. The postmaster can take several command-line arguments to tune its behavior. However, supplying arguments is necessary only if you intend to run multiple sites or a non-default site.

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

Notation

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

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

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

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

> Note: At the time of writing (Postgres v6.4) the notation for flagging commands is not universally consistant throughout the documentation set. Please report problems to the Documentation Mailing List.

Y2K Statement

Author: Written by Thomas Lockhart on 1998-10-22.

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

> The author of this statement, a volunteer on the Postgres support team since November, 1996, is not aware of any problems in the Postgres code base related to time transitions around Jan 1, 2000 (Y2K).
>
> The author of this statement is not aware of any reports of Y2K problems uncovered in regression testing or in other field use of recent or current versions of Postgres. We might have expected to hear about problems if they existed, given the installed base and the active participation of users on the support mailing lists.
>
> To the best of the author's knowledge, the assumptions Postgres makes about dates specified with a two-digit year are documented in the current User's Guide in the chapter on data types. For two-digit years, the significant transition year is 1970, not 2000; e.g. 70-01-01 is interpreted as 1970-01-01 , whereas 69-01-01 is interpreted as 2069-01-01 .
>
> Any Y2K problems in the underlying OS related to obtaining "the current time" may propagate into apparent Y2K problems in Postgres.

Refer to The Gnu Project and The Perl Institute for further discussion of Y2K issues, particularly as it relates to open source, no fee software.

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. SQL Syntax

Key Words

SQL92 defines key words for the language which have specific meaning. Some key words are reserved, which indicates that they are restricted to appear in only certain contexts. Other key words are not restricted, which indicates that in certain contexts they have a specific meaning but are not otherwise constrained.

Postgres implements an extended subset of the SQL92 and SQL3 languages. Some language elements are not as restricted in this implementation as is called for in the language standards, in part due to the extensibility features of Postgres.

Information on SQL92 and SQL3 key words is derived from *A Guide to the SQL Standard*.

Reserved Key Words

SQL92 and SQL3 have reserved key words which are not allowed as identifiers and not allowed in any usage other than as fundamental tokens in SQL statements. Postgres has additional key words which have similar restrictions. In particular, these key words are not allowed as column or table names, though in some cases they are allowed to be column labels (i.e. in AS clauses).

> Tip: Any string can be specified as an identifier if surrounded by double quotes (like this!). Some care is required since such an identifier will be case sensitive and will retain embedded whitespace other special characters.

The following are Postgres reserved words which are neither SQL92 nor SQL3 reserved words. These are allowed to be present as column labels, but not as identifiers:

```
ABORT ANALYZE
BINARY
CLUSTER CONSTRAINT COPY
DO
EXPLAIN EXTEND
LISTEN LOAD LOCK
MOVE
NEW NONE NOTIFY
RESET
SETOF SHOW
UNLISTEN UNTIL
VACUUM VERBOSE
```

The following are Postgres reserved words which are also SQL92 or SQL3 reserved words, and which are allowed to be present as column labels, but not as identifiers:

```
CROSS CURRENT
FALSE FOREIGN
GROUP
ORDER
POSITION PRECISION
TABLE TRANSACTION TRUE
```

The following are Postgres reserved words which are also SQL92 or SQL3 reserved words:

```
ADD ALL ALTER AND ANY AS ASC
BEGIN BETWEEN BOTH BY
CASCADE CAST CHAR CHARACTER CHECK CLOSE COLLATE COLUMN COMMIT
CONSTRAINT CREATE CURRENT_DATE CURRENT_TIME
CURRENT_TIMESTAMP CURRENT_USER CURSOR
DECIMAL DECLARE DEFAULT DELETE DESC DISTINCT DROP
END EXECUTE EXISTS EXTRACT
FETCH FLOAT FOR FROM FULL
GRANT
HAVING
IN INNER INSERT INTERVAL INTO IS
JOIN
LEADING LEFT LIKE LOCAL
NAMES NATIONAL NATURAL NCHAR NO NOT NULL NUMERIC
ON OR OUTER
PARTIAL PRIMARY PRIVILEGES PROCEDURE PUBLIC
REFERENCES REVOKE RIGHT ROLLBACK
SELECT SET SUBSTRING
TO TRAILING TRIM
UNION UNIQUE UPDATE USER USING
VALUES VARCHAR VARYING VIEW
WHERE WITH WORK
```

The following are SQL92 reserved key words which are not Postgres reserved key words, but which if used as function names are always translated into the function length:

```
CHAR_LENGTH CHARACTER_LENGTH
```

The following are SQL92 or SQL3 reserved key words which are not Postgres reserved key words, but if used as type names which are always translated into an alternate, native type:

```
BOOLEAN DOUBLE FLOAT INT INTEGER INTERVAL REAL SMALLINT
```

The following are either SQL92 or SQL3 reserved key words which are not key words in Postgres. These have no proscribed usage in Postgres at the time of writing (v6.4) but may become reserved key words in the future:

> Note: Some of these key words represent functions in SQL92. These functions are defined in Postgres, but the parser does not consider the names to be key words and they are allowed in other contexts.

```
ALLOCATE ARE ASSERTION AT AUTHORIZATION AVG
BIT BIT_LENGTH
CASCADED CASE CATALOG COALESCE COLLATION
CONNECT CONNECTION CONSTRAINTS CONTINUE CONVERT CORRESPONDING COUNT
DATE DEALLOCATE DEC DESCRIBE DESCRIPTOR DIAGNOSTICS DISCONNECT DOMAIN
ELSE END-EXEC ESCAPE EXCEPT EXCEPTION EXEC EXTERNAL
FIRST FOUND
GET GLOBAL GO GOTO
IDENTITY IMMEDIATE INDICATOR INITIALLY INPUT INTERSECT ISOLATION
LAST LEVEL LOWER
MAX MIN MODULE
NULLIF
OCTET_LENGTH OPEN OUTPUT OVERLAPS
PREPARE PRESERVE
RESTRICT ROWS
SCHEMA SECTION SESSION SESSION_USER SIZE SOME
SQL SQLCODE SQLERROR SQLSTATE SUM SYSTEM_USER
TEMPORARY THEN TRANSLATE TRANSLATION
UNKNOWN UPPER USAGE
VALUE
WHEN WHENEVER WRITE
```

Non-reserved Keywords

SQL92 and SQL3 have non-reserved keywords which have a proscribed meaning in the language but which are also allowed as identifiers. Postgres has additional keywords which allow similar unrestricted usage. In particular, these keywords are allowed as column or table names.

The following are Postgres non-reserved key words which are neither SQL92 nor SQL3 non-reserved key words:

```
AFTER AGGREGATE
BACKWARD BEFORE
CACHE CREATEDB CREATEUSER CYCLE
DATABASE DELIMITERS
EACH ENCODING
FORWARD FUNCTION
HANDLER
INCREMENT INDEX INHERITS INSENSITIVE INSTEAD ISNULL
LANCOMPILER LOCATION
MAXVALUE MINVALUE
NOCREATEDB NOCREATEUSER NOTHING NOTNULL
OIDS OPERATOR
PASSWORD PROCEDURAL
RECIPE RENAME  RETURNS ROW RULE
SEQUENCE SERIAL START STATEMENT STDIN STDOUT
TRUSTED
VALID VERSION
```

The following are Postgres non-reserved key words which are SQL92 or SQL3 reserved key words:

```
ABSOLUTE ACTION
DAY
HOUR
INSENSITIVE
KEY
LANGUAGE
MATCH MINUTE MONTH
NEXT
OF ONLY OPTION
PRIOR PRIVILEGES
READ RELATIVE
SCROLL SECOND
TIME TIMESTAMP TIMEZONE_HOUR TIMEZONE_MINUTE TRIGGER
YEAR
ZONE
```

The following are Postgres non-reserved key words which are also either SQL92 or SQL3 non-reserved key words:

```
TYPE
```

The following are either SQL92 or SQL3 non-reserved key words which are not key words of any kind in Postgres:

```
ADA
C CATALOG_NAME CHARACTER_SET_CATALOG CHARACTER_SET_NAME
CHARACTER_SET_SCHEMA CLASS_ORIGIN COBOL COLLATION_CATALOG
COLLATION_NAME COLLATION_SCHEMA COLUMN_NAME
COMMAND_FUNCTION COMMITTED CONDITION_NUMBER
CONNECTION_NAME CONSTRAINT_CATALOG CONSTRAINT_NAME
CONSTRAINT_SCHEMA CURSOR_NAME
DATA DATE_TIME_INTERVAL_CODE DATE_TIME_INTERVAL_PRECISION
DYNAMIC_FUNCTION
FORTRAN
LENGTH
MESSAGE_LENGTH MESSAGE_OCTET_LENGTH MORE MUMPS
NAME NULLABLE NUMBER
PAD PASCAL PLI
REPEATABLE RETURNED_LENGTH RETURNED_OCTET_LENGTH
RETURNED_SQLSTATE ROW_COUNT
SCALE SCHEMA_NAME SERIALIZABLE SERVER_NAME SPACE
SUBCLASS_ORIGIN
TABLE_NAME
UNCOMMITTED UNNAMED
```

Chapter 5. Data Types

Describes the built-in data types available in Postgres.

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.

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.

Table 5-1. Postgres Data Types

Postgres Type	SQL92 or SQL3 Type	Description
bool	boolean	logical boolean (true/false)
box		rectangular box in 2D plane
char(n)	character(n)	fixed-length character string
cidr		IP version 4 network or host address
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
inet		IP version 4 network or host address
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
int8		signed 8-byte integer
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
serial		unique id for indexing and cross-reference
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

Note: The cidr and inet types are designed to handle any IP type but only ipv4 is handled in the current implementation. Everything here that talks about ipv4 will apply to ipv6 in a future release.

Table 5-2. Postgres Function Constants

Postgres Function	SQL92 Constant	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 aspects of the standard which are ill considered and which should not live through subsequent standards. Postgres will not make great efforts to conform to these features; however, these tend to apply in little-used or obsure cases, and a typical user is not likely to run into them.

Most of the input and output functions corresponding to the base types (e.g., integers and floating point numbers) do some error-checking. Some of the operators and functions (e.g., addition and multiplication) do not perform run-time error-checking in the interests of improving execution speed. On some systems, for example, the numeric operators for some data types may silently underflow or overflow.

Note that 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.

Note: The original Postgres v4.2 code received from Berkeley rounded all double precision floating point results to six digits for output. Starting with v6.1, floating point numbers are allowed to retain most of the intrinsic precision of the type (typically 15 digits for doubles, 6 digits for 4-byte floats). Other types with underlying floating point fields (e.g. geometric types) carry similar precision.

Numeric Types

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

Table 5-3. Postgres Numeric Types

Numeric Type	Storage	Description	Range
float4	4 bytes	Variable-precision	6 decimal places
float8	8 bytes	Variable-precision	15 decimal places
int2	2 bytes	Fixed-precision	-32768 to +32767
int4	4 bytes	Usual choice for fixed-precision	-2147483648 to +2147483647
int8	8 bytes	Very large range fixed-precision	+/- > 18 decimal places
serial	4 bytes	Identifer or cross-reference	0 to +2147483647

The numeric types have a full set of corresponding arithmetic operators and functions. Refer to *Numerical Operators* and *Mathematical Functions* for more information.

The serial type is a special-case type constructed by Postgres from other existing components. It is typically used to create unique identifiers for table entries. In the current implementation, specifying

```
CREATE TABLE tablename (colname SERIAL);
```

is equivalent to specifying:

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename
    (colname INT4 DEFAULT nextval('tablename_colname_seq');
CREATE UNIQUE INDEX tablename_colname_key on tablename (colname);
```

Caution

The implicit sequence created for the serial type will not be automatically removed when the table is dropped. So, the following commands executed in order will likely fail:

```
CREATE TABLE tablename (colname SERIAL);
DROP TABLE tablename;
CREATE TABLE tablename (colname SERIAL);
```

The sequence will remain in the database until explicitly dropped using DROP SEQUENCE.

The exact numerics decimal and numeric have fully implemented syntax but currently (Postgres v6.4) support only a small range of precision and/or range values. The int8 type may not be available on all platforms since it relies on compiler support for this.

Monetary Type

The money type supports US-style currency with fixed decimal point representation. If Postgres is compiled with USE_LOCALE then the money type should use the monetary conventions defined for locale(7).

Table 5-4. Postgres Monetary Types

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

numeric should eventually replace the money type. It has a fully implemented syntax but currently (Postgres v6.4) support only a small range of precision and/or range values and cannot adequately substitute for the money type.

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 5-5. 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 is one other fixed-length character type. The name type only has one purpose and that is to provide Postgres with a special type to use for internal names. It is not intended for use by the general user. It's length is currently defined as 32 chars but should be reference using NAMEDATALEN. This is set at compile time and may change in a future release.

Table 5-6. Postgres Specialty Character Type

Character Type	Storage	Description
name	32 bytes	Thirty-two character internal type

Date/Time Types

There are two fundamental kinds of date and time measurements: absolute clock times and relative time intervals. Both quantities should demonstrate 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 timestamp, interval, date and time.

In a future release, datetime and timespan are likely to merge with the SQL92 types timestamp, interval. Other date and time types are also available, mostly for historical reasons.

Table 5-7. 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

timestamp is currently implemented separately from datetime, although they share input and output routines.

Table 5-8. Postgres Date/Time Ranges

Date/Time Type	Earliest	Latest	Resolution
abstime	1901-12-14	2038-01-19	1 sec
date	4713 BC	32767 AD	1 day
datetime	4713 BC	1465001 AD	1 microsec to 14 digits
interval	-178000000 years	178000000 years	1 microsec
reltime	-68 years	+68 years	1 sec
time	00:00:00.00	23:59:59.99	1 microsec
timespan	-178000000 years	178000000 years	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. Two obvious problems are:

Although the date type does not have an associated time zone, the time type can or does.

The default time zone is specified as a constant integer 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 may vary through the year with daylight savings time boundaries.

To address 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 5-9. 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 5-10. 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 v6.4 (and earlier) the default date/time style is "non-European 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.
The SQL command SET TIME ZONE 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 5-11. 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.

> Historical Note: 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 5-12. 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 5-13. 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.

IP Version 4 Networks and Host Addresses

The cidr type stores networks specified in CIDR notation. The inet type stores hosts and networks in CIDR notation.

Table 5-14. PostgresIP Version 4 Types

IPV4 Type	Storage	Description	Range
cidr	variable	CIDR networks	Valid IPV4 CIDR blocks
inet	variable	nets and hosts	Valid IPV4 CIDR blocks

CIDR for IP Networks

The cidr type holds a CIDR network. The format for specifying networks is "x.x.x.x/y" where "x.x.x.x" is the network and "/y" is the number of bits in the netmask. If the "/y" part is left off, it is calculated using assumptions from the old class system except that it is extended to include at least all of the octets in the input. Here are some examples:

Table 5-15. PostgresIP Types Examples

CIDR Input	CIDR Displayed
192.168.1	192.168.1/24
192.168	192.168.0/24
128.1	128.1/16
128	128.0/16
128.1.2	128.1.2/24
10.1.2	10.1.2/24
10.1	10.1/16
10	10/8

inet for IP Networks

The inet type is designed to hold, in one field, all of the information about a host including the CIDR style subnet that it is in. Note that if you want to store proper CIDR networks, see the cidr type. The inet type is similar to the cidr type except that the bits in the host part can be non-zero. Functions exist to extract the various elements of the field.

The input format for this function is "x.x.x.x/y" where "x.x.x.x" is an internet host and y is the number of bits in the netmask. If the "/y" part is left off, it is treated as "/32." On output, the "/y" part is not printed if it is /32. This allows the type to be used as a straight host type by just leaving of the bits part.

Chapter 6. Operators

Describes the built-in operators available in Postgres.

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)
```

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 command (\dd) to show these operators.

Lexical Precedence

Operators have a precedence which is currently hardcoded into the parser. Most operators have the same precedence and are non-associative. This may lead to non-intuitive behavior; for example the boolean operators "<" and ">" have a different precedence that the boolean operators "<=" and ">=".

Table 6-1. Operator Ordering (decreasing precedence)

Element	Precedence	Description
UNION	left	SQL select construct
::		Postgres typecasting
[ ]	left	array delimiters
.	left	table/column delimiter
-	right	unary minus
;	left	statement termination, logarithm
:	right	exponentiation
|	left	start of interval
* /	left	multiplication, division
+ -	left	addition, subtraction
IS		test for TRUE, FALSE, NULL
ISNULL		test for NULL
NOTNULL		test for NOT NULL
(all other operators)		native and user-defined
IN		set membership
BETWEEN		containment
LIKE		string pattern matching
< >		boolean inequality
=	right	equality
NOT	right	negation
AND	left	logical intersection
OR	left	logical union

General Operators

The operators listed here are defined for a number of native data types, ranging from numeric types to data/time types.

Table 6-2. 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,marc,hermit’ ~~ ’%scrappy%’
!~~	NOT LIKE	’bruce’ !~~ ’%al%’
~	Match (regex), case sensitive	’thomas’ ~ ’.*thomas.*’
~*	Match (regex), case insensitive	’thomas’ ~* ’.*Thomas.*’
!~	Does not match (regex), case sensitive	’thomas’ !~ ’.*Thomas.*’
!~*	Does not match (regex), case insensitive	’thomas’ !~ ’.*vadim.*’

Numerical Operators

Table 6-3. 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

Geometric Operators

Table 6-4. 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 in polygon	# '((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

Time Interval Operators

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 6-5. 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?	

IP V4 Operators

Table 6-6. PostgresIP V4 CIDR Operators

Operator	Description	Usage
<	Less than	’192.168.1.5’::cidr < ’192.168.1.6’::cidr
<=	Less than or equal	’192.168.1.5’::cidr <= ’192.168.1.5’::cidr
=	Equals	’192.168.1.5’::cidr = ’192.168.1.5’::cidr
%gt;=	Greater or equal	’192.168.1.5’::cidr >= ’192.168.1.5’::cidr
%gt;	Greater	’192.168.1.5’::cidr %gt; ’192.168.1.4’::cidr
<>	Not equal	’192.168.1.5’::cidr <> ’192.168.1.4’::cidr
<<	is contained within	’192.168.1.5’::cidr << ’192.168.1/24’::cidr
<<=	is contained within or equals	’192.168.1/24’::cidr <<= ’192.168.1/24’::cidr
>>	contains	’192.168.1/24’::cidr >> ’192.168.1.5’::cidr
>>=	contains or equals	’192.168.1/24’::cidr >>= ’192.168.1/24’::cidr

Chapter 7. Functions

Describes the built-in functions available in Postgres.

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

Mathematical Functions

Table 7-1. Mathematical Functions

Function	Returns	Description	Example
dexp(float8)	float8	raise e to the specified exponent	dexp(2.0)
dpow(float8,float8)	float8	raise a number to the specified exponent	dpow(2.0, 16.0)
float(int)	float8	convert integer to floating point	float(2)
float4(int)	float4	convert integer to floating point	float4(2)
integer(float)	int	convert floating point to integer	integer(2.0)

String Functions

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

Table 7-2. SQL92 String Functions

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

Many string functions are available for text, varchar(), and char() types. Some are used internally to implement the SQL92 string functions listed above.

Table 7-3. String Functions

Function	Returns	Description	Example
char(text)	char	convert text to char type	char('text string')
char(varchar)	char	convert varchar to char type	char(varchar 'varchar string')
initcap(text)	text	first letter of each word to upper case	initcap('thomas')
lower(text)	text	convert text to lower case	lower('TOM')
lpad(text,int,text)	text	left pad string to specified 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 specified length	rpad('hi',4,'x')
rtrim(text,text)	text	right trim characters from text	rtrim('trimxxxx','x')
substr(text,int[,int])	text	extract specified substring	substr('hi there',3,5)
text(char)	text	convert char to text type	text('char string')
text(varchar)	text	convert varchar to text type	text(varchar 'varchar string')
translate(text,from,to)	text	convert character in string	translate('12345', '1', 'a')
varchar(char)	varchar	convert char to varchar type	varchar('char string')
varchar(text)	varchar	convert text to varchar type	varchar('text string')
upper(text)	text	convert text to upper case	upper('tom')

Most functions explicitly defined for text will work for char() and varchar() arguments.

Date/Time Functions

The date/time functions provide a powerful set of tools for manipulating various date/time types.

Table 7-4. Date/Time Functions

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

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).

Geometric Functions

The geometric types point, box, lseg, line, path, polygon, and circle have a large set of native support functions.

Table 7-5. Geometric Functions

Function	Returns	Description	Example
area(box)	float8	area of box	area('((0,0),(1,1))'::box)
area(circle)	float8	area of circle	area('((0,0),2.0)'::circle)
box(box,box)	box	boxes to intersection box	box('((0,0),(1,1))','((-0.5,0.5),(2,2))')
center(box)	point	center of object	center('((0,0),(1,2))'::box)
center(circle)	point	center of object	center('((0,0),2.0)'::circle)
diameter(circle)	float8	diameter of circle	diameter('((0,0),2.0)'::circle)
height(box)	float8	vertical size of box	height('((0,0),(1,1))'::box)
isclosed(path)	bool	TRUE if this is a closed path	isclosed('((0,0),(1,1),(2,0))'::path)
isopen(path)	bool	TRUE if this is an open path	isopen('[(0,0),(1,1),(2,0)]'::path)
length(lseg)	float8	length of line segment	length('((-1,0),(1,0))'::lseg)
length(path)	float8	length of path	length('((0,0),(1,1),(2,0))'::path)
pclose(path)	path	convert path to closed variant	popen('[(0,0),(1,1),(2,0)]'::path)
point(lseg,lseg)	point	convert to point (intersection)	point('((-1,0),(1,0))'::lseg,'((-2,-2),(2,2))'::lseg)
points(path)	int4	number of points in path	points('[(0,0),(1,1),(2,0)]'::path)
popen(path)	path	convert path to open variant	popen('((0,0),(1,1),(2,0))'::path)
radius(circle)	float8	radius of circle	radius('((0,0),2.0)'::circle)
width(box)	float8	horizontal size of box	width('((0,0),(1,1))'::box)

Table 7-6. Geometric Type Conversion Functions

Function	Returns	Description	Example
box(circle)	box	convert circle to box	box('((0,0),2.0)'::circle)
box(point,point)	box	convert points to box	box('(0,0)'::point,'(1,-1)'::point)
box(polygon)	box	convert polygon to box	box('((0,0),(1,1),(2,0))'::polygon)
circle(box)	circle	convert to circle	circle('((0,0),(1,1))'::box)
circle(point,float8)	circle	convert to circle	circle('(0,0)'::point,2.0)
lseg(box)	lseg	convert diagonal to lseg	lseg('((-1,0),(1,0))'::box)
lseg(point,point)	lseg	convert to lseg	lseg('(-1,0)'::point,'(-1,0)'::point)
path(polygon)	point	convert to path	path('((0,0),(1,1),(2,0))'::polygon)
point(circle)	point	convert to point (center)	point('((0,0),2.0)'::circle)
point(lseg,lseg)	point	convert to point (intersection)	point('((-1,0),(1,0))'::lseg,'((-2,-2),(2,2))'::lseg)
point(polygon)	point	center of polygon	point('((0,0),(1,1),(2,0))'::polygon)
polygon(box)	polygon	convert to polygon with 12 points	polygon('((0,0),(1,1))'::box)
polygon(circle)	polygon	convert to polygon with 12 points	polygon('((0,0),2.0)'::circle)
polygon(npts,circle)	polygon	convert to polygon with npts points	polygon(12,'((0,0),2.0)'::circle)
polygon(path)	polygon	convert to polygon	polygon('((0,0),(1,1),(2,0))'::path)

Table 7-7. Geometric Upgrade Functions

Function	Returns	Description	Example
isoldpath(path)	path	test path for pre-v6.1 form	isoldpath('(1,3,0,0,1,-1,2,0)'::path)
revertpoly(polygon)	polygon	convert pre-v6.1 polygon	revertpoly('((0,0),(1,-1),(2,0))'::polygon)
upgradepath(path)	path	convert pre-v6.1 path	upgradepath('(1,3,0,0-,1,1,2,0)'::path)
upgradepoly(polygon)	polygon	convert pre-v6.1 polygon	upgradepoly('(0,1,2,0-,1,0)'::polygon)

IP V4 Functions

Table 7-8. PostgresIP V4 Functions

Function	Returns	Description	Example
broadcast(cidr)	text	construct broadcast address as text	broadcast('192.168.1.-5/24') ==> '192.168.1.255'
broadcast(inet)	text	construct broadcast address as text	broadcast('192.168.1.-5/24') ==> '192.168.1.255'
host(inet)	text	extract host address as text	host('192.168.1.5/24') ==> '192.168.1.5'
masklen(cidr)	int4	calculate netmask length	masklen('192.168.1.5-/24') ==> 24
masklen(inet)	int4	calculate netmask length	masklen('192.168.1.5-/24') ==> 24
netmask(inet)	text	construct netmask as text	netmask('192.168.1.5-/24') ==> '255.255.255.0'

Chapter 8. Type Conversion

SQL queries can, intentionally or not, require mixing of different data types in the same expression. Postgres has extensive facilities for evaluating mixed-type expressions.

In many cases a user will not need to understand the details of the type conversion mechanism. However, the implicit conversions done by Postgres can affect the apparent results of a query, and these results can be tailored by a user or programmer using explicit type coersion.

This chapter introduces the Postgres type conversion mechanisms and conventions. Refer to the relevant sections in the User's Guide and Programmer's Guide for more information on specific data types and allowed functions and operators.

The Programmer's Guide has more details on the exact algorithms used for implicit type conversion and coersion.

Overview

SQL is a strongly typed language. That is, every data item has an associated data type which determines its behavior and allowed usage. Postgres has an extensible type system which is much more general and flexible than other RDBMS implementations. Hence, most type conversion behavior in Postgres should be governed by general rules rather than by ad-hoc heuristics to allow mixed-type expressions to be meaningful, even with user-defined types.

The Postgres scanner/parser decodes lexical elements into only five fundamental categories: integers, floats, strings, names, and keywords. Most extended types are first tokenized into strings. The SQL language definition allows specifying type names with strings, and this mechanism is used by Postgres to start the parser down the correct path. For example, the query

```
tgl=> SELECT text 'Origin' AS "Label", point '(0,0)' AS "Value";
Label |Value
------+-----
Origin|(0,0)
(1 row)
```

has two strings, of type text and point. If a type is not specified, then the placeholder type unknown is assigned initially, to be resolved in later stages as described below.

There are four fundamental SQL constructs requiring distinct type conversion rules in the Postgres parser:

Operators

> Postgres allows expressions with left- and right-unary (one argument) operators, as well as binary (two argument) operators.

Function calls

> Much of the Postgres type system is built around a rich set of functions. Function calls have one or more arguments which, for any specific query, must be matched to the functions available in the system catalog.

Query targets

> SQL INSERT statements place the results of query into a table. The expressions in the query must be matched up with, and perhaps converted to, the target columns of the insert.

UNION queries

Since all select results from a UNION SELECT statement must appear in a single set of columns, the types of each SELECT clause must be matched up and converted to a uniform set.

Many of the general type conversion rules use simple conventions built on the Postgres function and operator system tables. There are some heuristics included in the conversion rules to better support conventions for the SQL92 standard native types such as smallint, integer, and float.

The Postgres parser uses the convention that all type conversion functions take a single argument of the source type and are named with the same name as the target type. Any function meeting this criteria is considered to be a valid conversion function, and may be used by the parser as such. This simple assumption gives the parser the power to explore type conversion possibilities without hardcoding, allowing extended user-defined types to use these same features transparently.

An additional heuristic is provided in the parser to allow better guesses at proper behavior for SQL standard types. There are five categories of types defined: boolean, string, numeric, geometric, and user-defined. Each category, with the exception of user-defined, has a "preferred type" which is used to resolve ambiguities in candidates. Each "user-defined" type is its own "preferred type", so ambiguous expressions (those with multiple candidate parsing solutions) with only one user-defined type can resolve to a single best choice, while those with multiple user-defined types will remain ambiguous and throw an error.

Ambiguous expressions which have candidate solutions within only one type category are likely to resolve, while ambiguous expressions with candidates spanning multiple categories are likely to throw an error and ask for clarification from the user.

Guidelines

All type conversion rules are designed with several principles in mind:

Implicit conversions should never have suprising or unpredictable outcomes.
User-defined types, of which the parser has no apriori knowledge, should be "higher" in the type heirarchy. In mixed-type expressions, native types shall always be converted to a user-defined type (of course, only if conversion is necessary).
User-defined types are not related. Currently, Postgres does not have information available to it on relationships between types, other than hardcoded heuristics for built-in types and implicit relationships based on available functions in the catalog.
There should be no extra overhead from the parser or executor if a query does not need implicit type conversion. That is, if a query is well formulated and the types already match up, then the query should proceed without spending extra time in the parser and without introducing unnecessary implicit conversion functions into the query.

Additionally, if a query usually requires an implicit conversion for a function, and if then the user defines an explicit function with the correct argument types, the parser should use this new function and will no longer do the implicit conversion using the old function.

Operators

Conversion Procedure

Procedure 8.1. Operator Evaluation

1. **Check for an exact match in the pg_operator system catalog.**
    a. **If one argument of a binary operator is unknown, then assume it is the same type as the other argument.**
    b. **Reverse the arguments, and look for an exact match with an operator which points to itself as being commutative. If found, then reverse the arguments in the parse tree and use this operator.**
2. **Look for the best match.**
    a. **Make a list of all operators of the same name.**
    b. **If only one operator is in the list, use it if the input type can be coerced, and throw an error if the type cannot be coerced.**
    c. **Keep all operators with the most explicit matches for types. Keep all if there are no explicit matches and move to the next step. If only one candidate remains, use it if the type can be coerced.**
    d. **If any input arguments are "unknown", categorize the input candidates as boolean, numeric, string, geometric, or user-defined. If there is a mix of categories, or more than one user-defined type, throw an error because the correct choice cannot be deduced without more clues. If only one category is present, then assign the "preferred type" to the input column which had been previously "unknown".**
    e. **Choose the candidate with the most exact type matches, and which matches the "preferred type" for each column category from the previous step. If there is still more than one candidate, or if there are none, then throw an error.**

Examples

Exponentiation Operator

There is only one exponentiation operator defined in the catalog, and it takes float8 arguments. The scanner assigns an initial type of int4 to both arguments of this query expression:

```
tgl=> select 2 ^ 3 AS "Exp";
Exp
---
  8
(1 row)
```

So the parser does a type conversion on both operands and the query is equivalent to

```
tgl=> select float8(2) ^ float8(3) AS "Exp";
Exp
---
  8
(1 row)
```

or

```
tgl=> select 2.0 ^ 3.0 AS "Exp";
Exp
---
  8
(1 row)
```

> Note: This last form has the least overhead, since no functions are called to do implicit type conversion. This is not an issue for small queries, but may have an impact on the performance of queries involving large tables.

String Concatenation

A string-like syntax is used for working with string types as well as for working with complex extended types. Strings with unspecified type are matched with likely operator candidates.

One unspecified argument:

```
tgl=> SELECT text 'abc' || 'def' AS "Text and Unknown";
Text and Unknown
----------------
abcdef
(1 row)
```

In this case the parser looks to see if there is an operator taking text for both arguments. Since there is, it assumes that the second argument should be interpreted as of type text.

Concatenation on unspecified types:

```
tgl=> SELECT 'abc' || 'def' AS "Unspecified";
Unspecified
-----------
abcdef
(1 row)
```

In this case there is no initial hint for which type to use, since no types are specified in the query. So, the parser looks for all candidate operators and finds that all arguments for all the candidates are string types. It chooses the "preferred type" for strings, text, for this query.

> Note: If a user defines a new type and defines an operator || to work with it, then this query would no longer succeed as written. The parser would now have candidate types from two categories, and could not decide which to use.

Factorial

This example illustrates an interesting result. Traditionally, the factorial operator is defined for integers only. The Postgres operator catalog has only one entry for factorial, taking an integer operand. If given a non-integer numeric argument, Postgres will try to convert that argument to an integer for evaluation of the factorial.

```
tgl=> select (4.3 !);
?column?
--------
      24
(1 row)
```

> Note: Of course, this leads to a mathematically suspect result, since in principle the factorial of a non-integer is not defined. However, the role of a database is not to teach mathematics, but to be a tool for data manipulation. If a user chooses to take the factorial of a floating point number, Postgres will try to oblige.

Functions

Procedure 8.2. Function Evaluation

1. **Check for an exact match in the pg_proc system catalog.**
2. **Look for the best match.**
    a. **Make a list of all functions of the same name with the same number of arguments.**
    b. **If only one function is in the list, use it if the input types can be coerced, and throw an error if the types cannot be coerced.**
    c. **Keep all functions with the most explicit matches for types. Keep all if there are no explicit matches and move to the next step. If only one candidate remains, use it if the type can be coerced.**
    d. **If any input arguments are "unknown", categorize the input candidate arguments as boolean, numeric, string, geometric, or user-defined. If there is a mix of categories, or more than one user-defined type, throw an error because the correct choice cannot be deduced without more clues. If only one category is present, then assign the "preferred type" to the input column which had been previously "unknown".**
    e. **Choose the candidate with the most exact type matches, and which matches the "preferred type" for each column category from the previous step. If there is still more than one candidate, or if there are none, then throw an error.**

Examples

Factorial Function

There is only one factorial function defined in the pg_proc catalog. So the following query automatically converts the int2 argument to int4:

```
tgl=> select int4fac(int2 '4');
int4fac
-------
     24
(1 row)
```

and is actually transformed by the parser to

```
tgl=> select int4fac(int4(int2 '4'));
int4fac
-------
     24
(1 row)
```

Substring Function

There are two substr functions declared in pg_proc. However, only one takes two arguments, of types text and int4.

If called with a string constant of unspecified type, the type is matched up directly with the only candidate function type:

```
tgl=> select substr('1234', 3);
substr
------
    34
(1 row)
```

If the string is declared to be of type varchar, as might be the case if it comes from a table, then the parser will try to coerce it to become text:

```
tgl=> select substr(varchar '1234', 3);
substr
------
    34
(1 row)
```

which is transformed by the parser to become

```
tgl=> select substr(text(varchar '1234'), 3);
substr
------
    34
(1 row)
```

> Note: There are some heuristics in the parser to optimize the relationship between the char, varchar, and text types. For this case, substr is called directly with the varchar string rather than inserting an explicit conversion call.

And, if the function is called with an int4, the parser will try to convert that to text:

```
tgl=> select substr(1234, 3);
substr
------
    34
(1 row)
```

actually executes as

```
tgl=> select substr(text(1234), 3);
substr
------
    34
(1 row)
```

Query Targets

Procedure 8.3. Target Evaluation

1. **Check for an exact match with the target.**
2. **Try to coerce the expression directly to the target type if necessary.**
3. **If the target is a fixed-length type (e.g. char or varchar declared with a length) then try to find a sizing function of the same name as the type taking two arguments, the first the type name and the second an integer length.**

Examples

varchar Storage

For a target column declared as varchar(4) the following query ensures that the target is sized correctly:

```
tgl=> CREATE TABLE vv (v varchar(4));
CREATE
tgl=> INSERT INTO vv SELECT 'abc' || 'def';
INSERT 392905 1
tgl=> select * from vv;
v
----
abcd
(1 row)
```

UNION Queries

The UNION construct is somewhat different in that it must match up possibly dissimilar types to become a single result set.

Procedure 8.4. UNION Evaluation

1. **Check for identical types for all results.**
2. **Coerce each result from the UNION clauses to match the type of the first SELECT clause or the target column.**

Examples

Underspecified Types

```
tgl=> SELECT text 'a' AS "Text" UNION SELECT 'b';
Text
----
a
b
(2 rows)
```

Simple UNION

```
tgl=> SELECT 1.2 AS Float8 UNION SELECT 1;
Float8
------
     1
   1.2
(2 rows)
```

Transposed UNION

The types of the union are forced to match the types of the first/top clause in the union:

```
tgl=> SELECT 1 AS "All integers"
tgl-> UNION SELECT '2.2'::float4
tgl-> UNION SELECT 3.3;
All integers
------------
           1
           2
           3
(3 rows)
```

An alternate parser strategy could be to choose the "best" type of the bunch, but this is more difficult because of the nice recursion technique used in the parser. However, the "best" type is used when selecting into a table:

```
tgl=> CREATE TABLE ff (f float);
CREATE
tgl=> INSERT INTO ff
tgl-> SELECT 1
tgl-> UNION SELECT '2.2'::float4
tgl-> UNION SELECT 3.3;
INSERT 0 3
tgl=> SELECT f AS "Floating point" from ff;
  Floating point
----------------
               1
2.20000004768372
             3.3
(3 rows)
```

Chapter 9. Indices and Keys

Author: Written by Herouth Maoz

Editor's Note: This originally appeared on the mailing list in response to the question: "What is the difference between PRIMARY KEY and UNIQUE constraints?".

```
Subject: Re: [QUESTIONS] PRIMARY KEY | UNIQUE

        What's the difference between:

              PRIMARY KEY(fields,...) and
              UNIQUE (fields,...)

       - Is this an alias?
       - If PRIMARY KEY is already unique, then why
         is there another kind of key named UNIQUE?
```

A primary key is the field(s) used to identify a specific row. For example, Social Security numbers identifying a person.

A simply UNIQUE combination of fields has nothing to do with identifying the row. It's simply an integrity constraint. For example, I have collections of links. Each collection is identified by a unique number, which is the primary key. This key is used in relations.

However, my application requires that each collection will also have a unique name. Why? So that a human being who wants to modify a collection will be able to identify it. It's much harder to know, if you have two collections named "Life Science", the the one tagged 24433 is the one you need, and the one tagged 29882 is not.

So, the user selects the collection by its name. We therefore make sure, withing the database, that names are unique. However, no other table in the database relates to the collections table by the collection Name. That would be very inefficient.

Moreover, despite being unique, the collection name does not actually define the collection! For example, if somebody decided to change the name of the collection from "Life Science" to "Biology", it will still be the same collection, only with a different name. As long as the name is unique, that's OK.

So:

Primary key:

Is used for identifying the row and relating to it.
Is impossible (or hard) to update.
Should not allow NULLs.

Unique field(s):

Are used as an alternative access to the row.
Are updateable, so long as they are kept unique.
NULLs are acceptable.

As for why no non-unique keys are defined explicitly in standard SQL syntax? Well, you must understand that indices are implementation-dependent. SQL does not define the implementation, merely the relations between data in the database. Postgres does allow non-unique indices, but indices used to enforce SQL keys are always unique.

Thus, you may query a table by any combination of its columns, despite the fact that you don't have an index on these columns. The indexes are merely an implementational aid which each RDBMS offers you, in order to cause commonly used queries to be done more efficiently.

Some RDBMS may give you additional measures, such as keeping a key stored in main memory. They will have a special command, for example

```
CREATE MEMSTORE ON <table> COLUMNS <cols>
```

(this is not an existing command, just an example).

In fact, when you create a primary key or a unique combination of fields, nowhere in the SQL specification does it say that an index is created, nor that the retrieval of data by the key is going to be more efficient than a sequential scan!

So, if you want to use a combination of fields which is not unique as a secondary key, you really don’t have to specify anything - just start retrieving by that combination! However, if you want to make the retrieval efficient, you’ll have to resort to the means your RDBMS provider gives you - be it an index, my imaginary MEMSTORE command, or an intelligent RDBMS which creates indices without your knowledge based on the fact that you have sent it many queries based on a specific combination of keys... (It learns from experience).

Chapter 10. 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        text[][]
);
```

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 text (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 11. 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 12. 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 *Understanding the New SQL* or *A Guide to the SQL Standard*. You should also be aware that some features of Postgres 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. This is usually faster because the data is read (or written) as a single atomic transaction directly to or from the target table. An example would be:

```
COPY INTO weather FROM '/home/user/weather.txt'
    USING DELIMITERS '|';
```

where the path name for the source file must be available to the backend server machine, not just the client.

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 13. Disk Storage

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

Chapter 14. psql

This section needs to be written. Volunteers? In the meantime, refer to the information in the reference section of this manual.

Paging To Screen

Author: From Brett McCormick on the mailing list 1998-04-04.

To affect the paging behavior of your psql output, set or unset your PAGER environment variable. I always have to set mine before it will pause. And of course you have to do this before starting the program.

In csh/tcsh or other C shells:

```
unsetenv PAGER
```

while in sh/bash or other Bourne shells:

```
unset PAGER
```

Chapter 15. pgaccess

This section needs to be written. Volunteers?

Chapter 16. SQL Commands

This is reference information for the SQL commands supported by Postgres.

ABORT

Name

ABORT Aborts the current transaction

```
ABORT
```

Inputs

```
None.
```

Outputs

```
 ABORT
    Message returned if successful.
NOTICE: UserAbortTransactionBlock and not in in-progress state
ABORT
    If there is not any transaction currently in progress.
```

Description

ABORT rolls back the current transaction and causes all the updates made by the transaction to be discarded. This command is identical in behavior to the SQL92 command ROLLBACK, and is present only for historical reasons.

Notes

Use the COMMIT statement to successfully terminate a transaction.

Usage

```
--To abort all changes
--
ABORT WORK;
```

Compatibility

SQL92

This command is a Postgres extension present for historical reasons. ROLLBACK is the SQL92 equivalent command.

ALTER TABLE

Name

ALTER TABLE Modifies table properties

```
ALTER TABLE table
    [ * ] ADD [ COLUMN ] column type
ALTER TABLE table
    [ * ] RENAME [ COLUMN ] column TO newcolumn
ALTER TABLE table
    RENAME TO newtable
```

Inputs

table

The name of an existing table to alter.

column

Name of a new or existing column.

type

Type of the new column.

newcolumn

New name for an existing column.

newtable

New name for an existing column.

Outputs

ALTER

Message returned from column or table renaming.

NEW

Message returned from column addition.

ERROR

Message returned if table or column is not available.

Description

ALTER TABLE changes the definition of an existing table. The new columns and their types are specified in the same style and with the the same restrictions as in CREATE TABLE. The RENAME clause causes the name of a table or column to change without changing any of the

data contained in the affected table. Thus, the table or column will remain of the same type and size after this command is executed.

You must own the table in order to change its schema.

Notes

The keyword COLUMN is noise and can be omitted.

[*] following a name of a table indicates that statement should be run over that table and all tables below it in the inheritance hierarchy. The PostgreSQL User's Guide has further information on inheritance.

Refer to CREATE TABLE for a further description of valid arguments.

Usage

To add a column of type VARCHAR to a table:

```
ALTER TABLE distributors ADD COLUMN address VARCHAR(30);
```

To rename an existing column:

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

To rename an existing table:

```
ALTER TABLE distributors RENAME TO suppliers;
```

Compatibility

SQL92

ALTER TABLE/RENAME is a Postgres language extension.

SQL92 specifies some additional capabilities for ALTER TABLE statement which are not yet directly supported by Postgres:

```
ALTER TABLE table ALTER [ COLUMN ] column
    SET DEFAULT default

ALTER TABLE table ALTER [ COLUMN ] column
    ADD [ CONSTRAINT constraint ] table-constraint
```

Puts the default value or constraint specified into the definition of column in the table. See CREATE TABLE for the syntax of the default and table-constraint clauses. If a default clause already exists, it will be replaced by the new definition. If any constraints on this column already exist, they will be retained using a boolean AND with the new constraint.

Currently, to set new default constraints on an existing column the table must be recreated and reloaded:

```
CREATE TABLE temp AS SELECT * FROM distributors;
DROP TABLE distributors;
CREATE TABLE distributors (
    did      DECIMAL(3) DEFAULT 1,
    name     VARCHAR(40) NOT NULL,
    city     VARCHAR(30)
);
INSERT INTO distributors SELECT * FROM temp;
DROP TABLE temp;
```

```
ALTER TABLE table
    DROP DEFAULT default
ALTER TABLE table
    DROP CONSTRAINT constraint { RESTRICT | CASCADE }
```

Removes the default value specified by default or the rule specified by constraint from the definition of a table. If RESTRICT is specified only a constraint with no dependent constraints can be destroyed. If CASCADE is specified, Any constraints that are dependent on this constraint are also dropped.

Currently, to remove a default value or constraints on an existing column the table must be recreated and reloaded:

```
CREATE TABLE temp AS SELECT * FROM distributors;
DROP TABLE distributors;
CREATE TABLE distributors AS SELECT * FROM temp;
DROP TABLE temp;
```

```
ALTER TABLE table
    DROP [ COLUMN ] column { RESTRICT | CASCADE }
```

Removes a column from a table. If RESTRICT is specified only a column with no dependent objects can be destroyed. If CASCADE is specified, all objects that are dependent on this column are also dropped.

Currently, to remove an existing column the table must be recreated and reloaded:

```
CREATE TABLE temp AS SELECT did, city FROM distributors;
DROP TABLE distributors;
CREATE TABLE distributors (
    did      DECIMAL(3)  DEFAULT 1,
    name     VARCHAR(40) NOT NULL,
);
INSERT INTO distributors SELECT * FROM temp;
DROP TABLE temp;
```

ALTER USER

Name

ALTER USER Modifies user account information

```
ALTER USER username
    [ WITH PASSWORD password ]
    [ CREATEDB | NOCREATEDB ]
    [ CREATEUSER | NOCREATEUSER ]
    [ IN GROUP groupname [, ...] ]
    [ VALID UNTIL 'abstime' ]
```

Inputs

Refer to CREATE USER for a detailed description of each clause.

username

The Postgres account name of the user whose details are to be altered.

password

The new password to be used for this account.

groupname

The name of an access group into which this account is to be put.

abstime

The date (and, optionally, the time) at which this user's access is to be terminated.

Outputs

ALTER USER

Message returned if the alteration was successful.

ERROR: alterUser: user "username" does not exist

Error message returned if the user specified doesn't exist.

Description

ALTER USER is used to change the attributes of a user's Postgres account. Please note that it is not possible to alter a user's "usesysid" via the alter user statement. Also, it is only possible

for the Postgres user or any user with read and modify permissions on "pg_shadow" to alter user passwords.

If any of the clauses of the alter user statement are omitted, the corresponding value in the "pg_shadow" table is left unchanged.

Notes

ALTER USER statement is a Postgres language extension.

Refer to CREATE/DROP USER to create or remove a user account.

In the current release (v6.4), the IN GROUP clause is parsed but has no affect. When it is fully implemented, it is intended to modify the pg_group relation.

Usage

Change a user password

```
ALTER USER davide WITH PASSWORD hu8jmn3;
```

Change a user's valid until date

```
ALTER USER manuel VALID UNTIL 'Jan 31 2030';
```

Change a user's valid until date, specifying that his authorisation should expire at midday on 4th May 1998 using the time zone which is one hour ahead of UTC

```
ALTER USER chris VALID UNTIL 'May 4 12:00:00 1998 +1';
```

Give a user the ability to create other users and new databases.

```
ALTER USER miriam CREATEUSER CREATEDB;
```

Place a user in two groups

```
ALTER USER miriam IN GROUP sales, payroll;
```

Compatibility

SQL92

There is no ALTER USER statement in SQL92. The standard leaves the definition of users to the implementation.

BEGIN WORK

Name

BEGIN WORK Begins a transaction

```
BEGIN [ WORK | TRANSACTION ]
```

Inputs

None

Outputs

```
BEGIN
    This signifies that a new transaction has been started.
NOTICE: BeginTransactionBlock and not in default state
    This indicates that a transaction was already in progress.
   The current transaction is not affected.
```

Description

BEGIN initiates a user transaction which Postgres will guarantee is serializable with respect to all concurrently executing transactions. Postgres uses two-phase locking to perform this task. If the transaction is committed, Postgres will ensure either that all updates are done or else that none of them are done. Transactions have the standard ACID (atomic, consistent, isolatable, and durable) property.

Notes

The keyword TRANSACTION is just a cosmetic alternative to WORK. Neither keyword need be specified.

Refer to the LOCK statement for further information about locking tables inside a transaction.

Use COMMIT or ROLLBACK to terminate a transaction.

Usage

To begin a user transaction:

```
BEGIN WORK;
```

Compatibility

BEGIN is a Postgres language extension.

SQL92

There is no explicit BEGIN WORK command in SQL92; transaction initiation is always implicit and it terminates either with a COMMIT or with a ROLLBACK statement.

CLOSE

Name

CLOSE Close a cursor

```
CLOSE cursor
```

Inputs

```
cursor
    The name of an open cursor to close.
```

Outputs

```
CLOSE
    Message returned if the cursor is successfully closed.
NOTICE PerformPortalClose: portal "cursor" not found
    This warning is given if cursor is not declared or has
   already been closed.
```

Description

CLOSE frees the resources associated with an open cursor. After the cursor is closed, no subsequent operations are allowed on it. A cursor should be closed when it is no longer needed.

An implicit close is executed for every open cursor when a transaction is terminated by COMMIT or ROLLBACK.

Notes

Postgres does not have an explicit OPEN cursor statement; a cursor is considered open when it is declared. Use the DECLARE statement to declare a cursor.

Usage

Close the cursor liahona:

```
CLOSE liahona;
```

Compatibility

SQL92

CLOSE is fully compatible with SQL92.

CLUSTER

Name

CLUSTER Gives storage clustering advice to the backend

```
CLUSTER indexname ON table
```

Inputs

```
indexname
    The name of an index.
table
    The name of a table.
```

Outputs

```
CLUSTER
    The clustering was done successfully.
ERROR: relation <tablerelation_number> inherits "invoice"

ERROR: Relation x does not exist!
```

Description

CLUSTER instructs Postgres to cluster the class specified by classname approximately based on the index specified by indexname. The index must already have been defined on classname.

When a class is clustered, it is physically reordered based on the index information. The clustering is static. In other words, as the class is updated, the changes are not clustered. No attempt is made to keep new instances or updated tuples clustered. If one wishes, one can recluster manually by issuing the command again.

Notes

The table is actually copied to a temporary table in index order, then renamed back to the original name. For this reason, all grant permissions and other indexes are lost when clustering is performed.

In cases where you are accessing single rows randomly within a table, the actual order of the data in the heap table is unimportant. However, if you tend to access some data more than others, and there is an index that groups them together, you will benefit from using CLUSTER.

Another place CLUSTER is helpful is in cases where you use an index to pull out several rows from a table. If you are requesting a range of indexed values from a table, or a single indexed value that has multiple rows that match, CLUSTER will help because once the index identifies the heap page for the first row that matches, all other rows that match are probably already on the same heap page, saving disk accesses and speeding up the query.

There are two ways to cluster data. The first is with the CLUSTER command, which reorders the original table with the ordering of the index you specify. This can be slow on large tables because the rows are fetched from the heap in index order, and if the heap table is unordered,

the entries are on random pages, so there is one disk page retrieved for every row moved. Postgres has a cache, but the majority of a big table will not fit in the cache.

Another way to cluster data is to use

```
SELECT ... INTO TABLE temp FROM ... ORDER BY ...
```

This uses the Postgres sorting code in ORDER BY to match the index, and is much faster for unordered data. You then drop the old table, use ALTER TABLE/RENAME to rename temp to the old name, and recreate any indexes. The only problem is that OIDs will not be preserved. From then on, CLUSTER should be fast because most of the heap data has already been ordered, and the existing index is used.

Usage

Cluster the employees relation on the basis of its salary attribute

```
CLUSTER emp_ind ON emp
```

Compatibility

SQL92

There is no CLUSTER statement in SQL92.

COMMIT

Name

COMMIT Commits the current transaction

```
COMMIT [ WORK | TRANSACTION ]
```

Inputs

```
None
```

Outputs

```
END
   Message returned if the transaction is successfully
   committed.
NOTICE EndTransactionBlock and not inprogress/abort state
   If there is no transaction in progress.
```

Description

COMMIT commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

Notes

The keywords WORK and TRANSACTION are noise and can be omitted.

Use the ROLLBACK statement to abort a transaction.

Usage

To make all changes permanent:

```
COMMIT WORK;
```

Compatibility

SQL92

Full compatibility.

COPY

Name

COPY Copies data between files and tables

```
COPY [ BINARY ] table [ WITH OIDS ]
    FROM { 'filename' | stdin }
    [ USING DELIMITERS 'delimiter' ]
COPY [ BINARY ] table [ WITH OIDS ]
    TO { 'filename' | stdout }
    [ USING DELIMITERS 'delimiter' ]
```

Inputs

BINARY

Changes the behavior of field formatting, forcing all data to be stored or read as binary objects rather than as text.

table

The name of an existing table.

WITH OIDS

Copies the internal unique object id (OID) for each row.

filename

The absolute Unix pathname of the input or output file.

stdin

Specifies that input comes from a pipe or terminal.

stdout

Specifies that output goes to a pipe or terminal.

delimiter

A character that delimits the input or output fields.

Outputs

COPY

The copy completed successfully.

ERROR: error message

The copy failed for the reason stated in the error message.

Description

COPY moves data between Postgres tables and standard Unix files.

COPY instructs the Postgres backend to directly read from or write to a file. The file must be directly visible to the backend and the name must be specified from the viewpoint of the backend. If stdin or stdout are specified, data flows through the client frontend to the backend.

Notes

The BINARY keyword will force all data to be stored/read as binary objects rather than as text. It is somewhat faster than the normal copy command, but is not generally portable, and the files generated are somewhat larger, although this factor is highly dependent on the data itself. By

default, a text copy uses a tab ("\t") character as a delimiter. The delimiter may also be changed to any other single character with the keyword phrase USING DELIMITERS. Characters in data fields which happen to match the delimiter character will be quoted.

You must have select access on any table whose values are read by COPY, and either insert or update access to a table into which values are being inserted by COPY. The backend also needs appropriate Unix permissions for any file read or written by COPY.

The keyword phrase USING DELIMITERS specifies a single character to be used for all delimiters between columns. If multiple characters are specified in the delimiter string, only the first character is used.

Tip: Do not confuse COPY with the psql instruction \copy.

File Formats

Text Format

When COPY TO is used without the BINARY option, the file generated will have each row (instance) on a single line, with each column (attribute) separated by the delimiter character. Embedded delimiter characters will be preceded by a backslash character ("\"). The attribute values themselves are strings generated by the output function associated with each attribute type. The output function for a type should not try to generate the backslash character; this will be handled by COPY itself.

The actual format for each instance is

```
<attr1><separator><attr2><separator>...<separator><attrn><newline>
```

The oid is placed on the beginning of the line if WITH OIDS is specified.

If COPY is sending its output to standard output instead of a file, it will send a backslash("\") and a period (".") followed immediately by a newline, on a separate line, when it is done. Similarly, if COPY is reading from standard input, it will expect a backslash ("\") and a period (".") followed by a newline, as the first three characters on a line to denote end-of-file. However, COPY will terminate (followed by the backend itself) if a true EOF is encountered before this special end-of-file pattern is found.

The backslash character has other special meanings. NULL attributes are output as "\N". A literal backslash character is output as two consecutive backslashes ("\\"). A literal tab character is represented as a backslash and a tab. A literal newline character is represented as a backslash and a newline. When loading text data not generated by Postgres, you will need to convert backslash characters ("\") to double-backslashes ("\\") to ensure that they are loaded properly.

Binary Format

In the case of COPY BINARY, the first four bytes in the file will be the number of instances in the file. If this number is zero, the COPY BINARY command will read until end of file is encountered. Otherwise, it will stop reading when this number of instances has been read. Remaining data in the file will be ignored.

The format for each instance in the file is as follows. Note that this format must be followed exactly. Unsigned four-byte integer quantities are called uint32 in the table below.

Table 16-1. Contents of a binary copy file

At the start of the file	
uint32	number of tuples
For each tuple	
uint32	total length of tuple data
uint32	oid (if specified)
uint32	number of null attributes
[uint32,...,uint32]	attribute numbers of attributes, counting from 0
-	<tuple data>

Alignment of Binary Data

On Sun-3s, 2-byte attributes are aligned on two-byte boundaries, and all larger attributes are aligned on four-byte boundaries. Character attributes are aligned on single-byte boundaries. On most other machines, all attributes larger than 1 byte are aligned on four-byte boundaries. Note that variable length attributes are preceded by the attribute's length; arrays are simply contiguous streams of the array element type.

Usage

The following example copies a table to standard output, using a vertical bar ("|") as the field delimiter:

```
COPY country TO stdout USING DELIMITERS '|';
```

To copy data from a Unix file into a table "country":

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

Here is a sample of data suitable for copying into a table from stdin (so it has the termination sequence on the last line):

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
...
ZM      ZAMBIA
ZW      ZIMBABWE
\.
```

The same data, output in binary format on a Linux/i586 machine. The data is shown after filtering through the Unix utility od -c. The table has three fields; the first is char(2) and the second is text. All the rows have a null value in the third field. Notice how the char(2) field is padded with nulls to four bytes and the text field is preceded by its length:

```
355  \0  \0  \0 027  \0  \0  \0 001  \0  \0  \0 002  \0  \0  \0
006  \0  \0  \0   A   F  \0  \0 017  \0  \0  \0   A   F   G   H
  A   N   I   S   T   A   N 023  \0  \0  \0 001  \0  \0  \0 002
 \0  \0  \0 006  \0  \0  \0   A   L  \0  \0  \v  \0  \0  \0   A
  L   B   A   N   I   A 023  \0  \0  \0 001  \0  \0  \0 002  \0
 \0  \0 006  \0  \0  \0   D   Z  \0  \0  \v  \0  \0  \0   A   L
  G   E   R   I   A
...                 \n  \0  \0  \0   Z   A   M   B   I   A 024  \0
 \0  \0 001  \0  \0  \0 002  \0  \0  \0 006  \0  \0  \0   Z   W
 \0  \0  \f  \0  \0  \0   Z   I   M   B   A   B   W   E
```

Bugs

COPY stops operation at the first error. This should not lead to problems in the event of a COPY FROM, but the target relation will, of course, be partially modified in a COPY TO. The VACUUM query should be used to clean up after a failed copy.

Because the Postgres backend's current working directory is not usually the same as the user's working directory, the result of copying to a file "foo" (without additional path information) may yield unexpected results for the naive user. In this case, foo will wind up in $PGDATA/foo. In general, the full pathname as it would appear to the backend server machine should be used when specifying files to be copied.

Files used as arguments to COPY must reside on or be accessible to the database server machine by being either on local disks or on a networked file system.

When a TCP/IP connection from one machine to another is used, and a target file is specified, the target file will be written on the machine where the backend is running rather than the user's machine.

Compatibility

SQL92

There is no COPY statement in SQL92.

CREATE AGGREGATE

Name

CREATE AGGREGATE Defines a new aggregate function

```
CREATE AGGREGATE name [ AS ]
    ( BASETYPE    = data_type
    [ , SFUNC1    = sfunc1
      , STYPE1    = sfunc1_return_type ]
    [ , SFUNC2    = sfunc2
      , STYPE2    = sfunc2_return_type ]
    [ , FINALFUNC = ffunc ]
    [ , INITCOND1 = initial_condition1 ]
    [ , INITCOND2 = initial_condition2 ]
    )
```

Inputs

name

The name of an aggregate function to create.

data_type

The fundamental data type on which this aggregate function operates.

sfunc1

The state transition function to be called for every non-NULL field from the source column. It takes a variable of type sfunc1_return_type as the first argument and that field as the second argument.

sfunc1_return_type

The return type of the first transition function.

sfunc2

The state transition function to be called for every non-NULL field from the source column. It takes a variable

of type sfunc2_return_type as the only argument and returns a variable of the same type.

sfunc2_return_type

The return type of the second transition function.

ffunc

The final function called after traversing all input fields. This function must take two arguments of types sfunc1_return_type and sfunc2_return_type.

initial_condition1

The initial value for the first transition function argument.

initial_condition2

The initial value for the second transition function argument.

Outputs

CREATE

Message returned if the command completes successfully.

Description

CREATE AGGREGATE allows a user or programmer to extend Postgres functionality by defining new aggregate functions. Some aggregate functions for base types such as min(int4) and avg(float8) are already provided in the base distribution. If one defines new types or needs an aggregate function not already provided then CREATE AGGREGATE can be used to provide the desired features.

An aggregate function can require up to three functions, two state transition functions, sfunc1 and sfunc2:

```
sfunc1( internal-state1, next-data_item ) ---> next-internal-state1
```

```
sfunc2( internal-state2 ) ---> next-internal-state2
```

and a final calculation function, ffunc:

```
ffunc(internal-state1, internal-state2) ---> aggregate-value
```

Postgres creates up to two temporary variables (referred to here as temp1 and temp2) to hold intermediate results used as arguments to the transition functions.

These transition functions are required to have the following properties:

The arguments to sfunc1 must be temp1 of type sfunc1_return_type and column_value of type data_type. The return value must be of type sfunc1_return_type and will be used as the first argument in the next call to sfunc1.

The argument and return value of sfunc2 must be temp2 of type sfunc2_return_type.

The arguments to the final-calculation-function must be temp1 and temp2 and its return value must be a Postgres base type (not necessarily data_type which had been specified for BASETYPE).

FINALFUNC should be specified if and only if both state-transition functions are specified.

An aggregate function may also require one or two initial conditions, one for each transition function. These are specified and stored in the database as fields of type text.

Notes

Use DROP AGGREGATE to drop aggregate functions.

It is possible to specify aggregate functions that have varying combinations of state and final functions. For example, the count aggregate requires SFUNC2 (an incrementing function) but not SFUNC1 or FINALFUNC, whereas the sum aggregate requires SFUNC1 (an addition function) but not SFUNC2 or FINALFUNC and the avg aggregate requires both of the above state functions as well as a FINALFUNC (a division function) to produce its answer. In any case, at least one state function must be defined, and any SFUNC2 must have a corresponding INITCOND2.

Usage

Refer to the chapter on aggregate functions in the PostgreSQL Programmer's Guide on aggregate functions for complete examples of usage.

Compatibility

SQL92

CREATE AGGREGATE is a Postgres language extension. There is no CREATE AGGREGATE in SQL92.

CREATE DATABASE

Name

CREATE DATABASE Creates a new database

```
CREATE DATABASE name [ WITH LOCATION = 'dbpath' ]
```

Inputs

name

The name of a database to create.

dbpath

An alternate location can be specified as either an environment variable known to the backend server (e.g. 'PGDATA2') or as an absolute path name (e.g. '/usr/local/pgsql/data'). In either case, the location must be pre-configured by initlocation.

Outputs

CREATEDB

Message returned if the command completes successfully.

WARN: createdb: database "name" already exists.

This occurs if database specified already exists.

ERROR: Unable to create database directory directory

There was a problem with creating the required directory; this operation will need permissions for the postgres user on the specified location.

Description

CREATE DATABASE creates a new Postgres database. The creator becomes the administrator of the new database.

Notes

CREATE DATABASE is a Postgres language extension.

Use DROP DATABASE to remove a database.

Usage

To create a new database:

```
olly=> create database lusiadas;
```

To create a new database in an alternate area ~/private_db:

```
$ mkdir private_db
$ initlocation ~/private_db
Creating Postgres database system directory /home/olly/private_db/base

$ psql olly
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

olly=> create database elsewhere with location =
'/home/olly/private_db';
    CREATEDB
```

Bugs

There are security and data integrity issues involved with using alternate database locations specified with absolute path names, and by default only an environment variable known to the backend may be specified for an alternate location. See the Administrator's Guide for more information.

Compatibility

SQL92

There is no CREATE DATABASE statement in SQL92.

The equivalent command in standard SQL is CREATE SCHEMA.

CREATE FUNCTION

Name

CREATE FUNCTION Defines a new function

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    AS path
    LANGUAGE 'langname'
```

Inputs

name

The name of a function to create.

ftype

The data type of function arguments.

rtype

The return data type.

path

May be either an SQL-query or an absolute path to an object file.

langname

may be 'C', 'sql', 'internal' or 'plname', where 'plname' is the name of a created procedural language. See CREATE LANGUAGE for details.

Outputs

CREATE

This is returned if the command completes successfully.

Description

CREATE FUNCTION allows a Postgres user to register a function with a database. Subsequently, this user is treated as the owner of the function.

Notes

Refer to the chapter on functions in the PostgreSQL Programmer's Guide for further information.

Use DROP FUNCTION to drop user-defined functions.

Usage

To create a simple SQL function:

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

SELECT one() AS answer;

answer
```

```
------
1
```

To create a C function, calling a routine from a user-created shared library. This particular routine calculates a check digit and returns TRUE if the check digit in the function parameters is correct. It is intended for use in a CHECK contraint.

```
CREATE FUNCTION ean_checkdigit(bpchar, bpchar) RETURNS bool
    AS '/usr1/proj/bray/sql/funcs.so' LANGUAGE 'c';

CREATE TABLE product
(
     id              char(8)         PRIMARY KEY,
     eanprefix       char(8)         CHECK (eanprefix ~
'[0-9]{2}-[0-9]{5}')
                                     REFERENCES brandname(ean_prefix),
     eancode         char(6)         CHECK (eancode ~ '[0-9]{6}'),
     CONSTRAINT ean CHECK (ean_checkdigit(eanprefix, eancode))
);
```

Bugs

A C function cannot return a set of values.

Compatibility

CREATE FUNCTION is a Postgres language extension.

SQL/PSM

> Note: PSM stands for Persistent Stored Modules. It is a procedural language and it was originally hoped that PSM would be ratified as an official standard by late 1996. As of mid-1998, this has not yet happened, but it is hoped that PSM will eventually become a standard.

SQL/PSM CREATE FUNCTION has the following syntax:

```
CREATE FUNCTION name
    ( [ [ IN | OUT | INOUT ] parm type [, ...] ] )
     RETURNS rtype
     LANGUAGE 'langname'
     ESPECIFIC routine
     SQL-statement
```

CREATE INDEX

Name

CREATE INDEX Constructs a secondary index

```
CREATE [ UNIQUE ] INDEX index_name
    ON table [ USING acc_name ]
    ( column [ ops_name] [, ...] )
CREATE [ UNIQUE ] INDEX index_name
    ON table [ USING acc_name ]
    ( func_name( column [, ... ]) ops_name )
```

Inputs

UNIQUE

Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update non-duplicate data will generate an error.

index_name

The name of the index to be created.

table

The name of the table to be indexed.

acc_name

the name of the access method which is to be used for the index. The default access method is BTREE. Postgres provides three access methods for secondary indexes:

BTREE

an implementation of the Lehman-Yao high-concurrency btrees.

RTREE

implements standard rtrees using Guttman’s quadratic split algorithm.

HASH

an implementation of Litwin’s linear hashing.

column

The name of a column of the table.

ops_name

An associated operator class. The following select list returns all ops_names:

```
SELECT am.amname AS acc_name,
       opc.opcname AS ops_name,
       opr.oprname AS ops_comp
    FROM pg_am am, pg_amop amop,
         pg_opclass opc, pg_operator opr
    WHERE amop.amopid = am.oid AND
          amop.amopclaid = opc.oid AND
          amop.amopopr = opr.oid
```

```
        ORDER BY acc_name, ops_name, ops_comp
```

`func_name`

A user-defined function, which returns a value that can be indexed.

Outputs

`CREATE`

The message returned if the index is successfully created.

`ERROR: Cannot create index: 'index_name' already exists.`

This error occurs if it is impossible to create the index.

Description

CREATE INDEX constructs an index index_name. on the specified table.

> Tip: Indexes are primarily used to enhance database performance. But inappropriate use will result in slower performance.

In the first syntax shown above, the key fields for the index are specified as column names; a column may also have an associated operator class. An operator class is used to specify the operators to be used for a particular index. For example, a btree index on four-byte integers would use the int4_ops class; this operator class includes comparison functions for four-byte integers. The default operator class is the appropriate operator class for that field type.

In the second syntax, an index is defined on the result of a user-defined function func_name applied to one or more attributes of a single class. These functional indexes can be used to obtain fast access to data based on operators that would normally require some transformation to apply them to the base data.

Notes

Currently, only the BTREE access method supports multi-column indexes. Up to 7 keys may be specified.

Use DROP INDEX to remove an index.

Usage

To create a btree index on the field title in the table films:

```
CREATE UNIQUE INDEX title_idx
    ON films (title);
```

Compatibility

SQL92

CREATE INDEX is a Postgres language extension.

There is no CREATE INDEX command in SQL92.

CREATE LANGUAGE

Name

CREATE LANGUAGE Defines a new language for functions

```
CREATE [ TRUSTED ] PROCEDURAL LANGUAGE 'langname'
    HANDLER call_handler
    LANCOMPILER 'comment'
```

Inputs

TRUSTED

TRUSTED specifies that the call handler for the language is safe; that is, it offers an unprivileged user no functionality to bypass access restrictions. If this keyword is omitted when registering the language, only users with the Postgres superuser privilege can use this language to create new functions (like the 'C' language).

langname

The name of the new procedural language. The language name is case insensitive. A procedural language cannot override one of the built-in languages of Postgres.

HANDLER call_handler

call_handler is the name of a previously registered function that will be called to execute the PL procedures.

comment

The LANCOMPILER argument is the string that will be inserted in the LANCOMPILER attribute of the new pg_language entry. At present, Postgres does not use this attribute in any way.

Outputs

CREATE

This message is returned if the language is successfully created.

ERROR: PL handler function funcname() doesn't exist

This error is returned if the function funcname() is not found.

Description

Using CREATE LANGUAGE, a Postgres user can register a new language with Postgres. Subsequently, functions and trigger procedures can be defined in this new language. The user must have the Postgres superuser privilege to register a new language.

Writing PL handlers

The call handler for a procedural language must be written in a compiler language such as 'C' and registered with Postgres as a function taking no arguments and returning the opaque type, a

placeholder for unspecified or undefined types.. This prevents the call handler from being called directly as a function from queries.

However, arguments must be supplied on the actual call when a PL function or trigger procedure in the language offered by the handler is to be executed.

When called from the trigger manager, the only argument is the object ID from the procedure's pg_proc entry. All other information from the trigger manager is found in the global CurrentTriggerData pointer.

When called from the function manager, the arguments are the object ID of the procedure's pg_proc entry, the number of arguments given to the PL function, the arguments in a FmgrValues structure and a pointer to a boolean where the function tells the caller if the return value is the SQL NULL value.

It's up to the call handler to fetch the pg_proc entry and to analyze the argument and return types of the called procedure. The AS clause from the CREATE FUNCTION of the procedure will be found in the prosrc attribute of the pg_proc table entry. This may be the source text in the procedural language itself (like for PL/Tcl), a pathname to a file or anything else that tells the call handler what to do in detail.

Notes

Use CREATE FUNCTION to create a function.

Use DROP LANGUAGE to drop procedural languages.

Refer to the table pg_language for further information:

```
Table    = pg_language
+--------------------------+--------------------------+-------+
|          Field           |          Type            | Length|
+--------------------------+--------------------------+-------+
| lanname                  | name                     |    32 |
| lancompiler              | text                     |   var |
+--------------------------+--------------------------+-------+

lanname |lancompiler
--------+--------------
internal|n/a
lisp    |/usr/ucb/liszt
C       |/bin/cc
sql     |postgres
```

Restrictions

Since the call handler for a procedural language must be registered with Postgres in the 'C' language, it inherits all the capabilities and restrictions of 'C' functions.

Bugs

At present, the definitions for a procedural language cannot be changed once they have been created.

Usage

This is a template for a PL handler written in 'C':

```
#include "executor/spi.h"
#include "commands/trigger.h"
#include "utils/elog.h"
#include "fmgr.h"        /* for FmgrValues struct */
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"
```

```
Datum
plsample_call_handler(
     Oid       prooid,
     int       pronargs,
     FmgrValues     *proargs,
     bool      *isNull)
{
     Datum          retval;
     TriggerData    *trigdata;

     if (CurrentTriggerData == NULL) {
          /*
           * Called as a function
           */

          retval = ...
     } else {
          /*
           * Called as a trigger procedure
           */
          trigdata = CurrentTriggerData;
          CurrentTriggerData = NULL;

          retval = ...
     }

     *isNull = false;
     return retval;
}
```

Only a few thousand lines of code have to be added instead of the dots to complete the PL call handler. See CREATE FUNCTION for information on how to compile it into a loadable module .

The following commands then register the sample procedural language:

```
CREATE FUNCTION plsample_call_handler () RETURNS opaque
    AS '/usr/local/pgsql/lib/plsample.so'
    LANGUAGE 'C';

CREATE PROCEDURAL LANGUAGE 'plsample'
    HANDLER plsample_call_handler
    LANCOMPILER 'PL/Sample';
```

Compatibility

CREATE LANGUAGE is a Postgres extension.

SQL92

There is no CREATE LANGUAGE statement in SQL92.

CREATE OPERATOR

Name

CREATE OPERATOR Defines a new user operator

```
CREATE OPERATOR name (
     PROCEDURE    = func_name
    [, LEFTARG    = type1 ]
    [, RIGHTARG   = type2 ]
    [, COMMUTATOR = com_op ]
    [, NEGATOR    = neg_op ]
    [, RESTRICT   = res_proc ]
    [, HASHES ]
    [, JOIN       = join_proc ]
    [, SORT       = sort_op [, ...] ]
)
```

Inputs

name

The operator to be defined. See below for allowable characters.

func_name

The function used to implement this operator.

type1

The type for the left-hand side of the operator, if any. This option would be omitted for a right-unary operator.

type2

The type for the right-hand side of the operator, if any. This option would be omitted for a left-unary operator.

com_op

The corresponding commutative operator.

neg_op

The corresponding negation operator.

res_proc

The corresponding restriction operator.

HASHES

This operator can support a hash-join algorithm.

join_proc

Procedure supporting table joins.

sort_op

Operator to use for sorting.

Outputs

CREATE

Message returned if the operator is successfully created.

Description

CREATE OPERATOR defines a new operator, name. The user who defines an operator

becomes its owner.

The operator name is a sequence of up to thirty two (32) characters in any combination from the following:

+ - * / < > = ~ ! @ # % ^ & | ` ? $:

> Note: No alphabetic characters are allowed in an operator name. This enables Postgres to parse SQL input into tokens without requiring spaces between each token.

The operator "!=" is mapped to "<>" on input, so they are therefore equivalent.

At least one of LEFTARG and RIGHTARG must be defined. For binary operators, both should be defined. For right unary operators, only LEFTARG should be defined, while for left unary operators only RIGHTARG should be defined.

Also, the func_name procedure must have been previously defined using CREATE FUNCTION and must be defined to accept the correct number of arguments (either one or two).

The commutator operator is present so that Postgres can reverse the order of the operands if it wishes. For example, the operator area-less-than, <<<, would have a commutator operator, area-greater-than, >>>. Hence, the query optimizer could freely convert:

```
"0,0,1,1"::box  >>> MYBOXES.description
```

to

```
MYBOXES.description <<< "0,0,1,1"::box
```

This allows the execution code to always use the latter representation and simplifies the query optimizer some what.

Suppose that an operator, area-equal, ===, exists, as well as an area not equal, !==. The negator operator allows the query optimizer to convert

```
NOT MYBOXES.description === "0,0,1,1"::box
```

to

```
MYBOXES.description !== "0,0,1,1"::box
```

If a commutator operator name is supplied, Postgres searches for it in the catalog. If it is found and it does not yet have a commutator itself, then the commutator's entry is updated to have the current (new) operator as its commutator. This applies to the negator, as well.

This is to allow the definition of two operators that are the commutators or the negators of each other. The first operator should be defined without a commutator or negator (as appropriate). When the second operator is defined, name the first as the commutator or negator. The first will be updated as a side effect.

The next two specifications are present to support the query optimizer in performing joins. Postgres can always evaluate a join (i.e., processing a clause with two tuple variables separated by an operator that returns a boolean) by iterative substitution [WONG76]. In addition, Postgres is planning on implementing a hash-join algorithm along the lines of [SHAP86]; however, it must know whether this strategy is applicable. For example, a hash-join algorithm is usable for a clause of the form:

```
MYBOXES.description === MYBOXES2.description
```

but not for a clause of the form:

```
MYBOXES.description <<< MYBOXES2.description.
```

The HASHES flag gives the needed information to the query optimizer concerning whether a hash join strategy is usable for the operator in question.

Similarly, the two sort operators indicate to the query optimizer whether merge-sort is a usable join strategy and what operators should be used to sort the two operand classes. For the === clause above, the optimizer must sort both relations using the operator, <<<. On the other hand, merge-sort is not usable with the clause:

```
MYBOXES.description <<< MYBOXES2.description
```

If other join strategies are found to be practical, Postgres will change the optimizer and run-time system to use them and will require additional specification when an operator is defined. Fortunately, the research community invents new join strategies infrequently, and the added generality of user-defined join strategies was not felt to be worth the complexity involved.

The last two pieces of the specification are present so the query optimizer can estimate result sizes. If a clause of the form:

```
MYBOXES.description <<< "0,0,1,1"::box
```

is present in the qualification, then Postgres may have to estimate the fraction of the instances in MYBOXES that satisfy the clause. The function res_proc must be a registered function (meaning it is already defined using define function(l)) which accepts one argument of the correct data type and returns a floating point number. The query optimizer simply calls this function, passing the parameter "0,0,1,1" and multiplies the result by the relation size to get the desired expected number of instances.

Similarly, when the operands of the operator both contain instance variables, the query optimizer must estimate the size of the resulting join. The function join_proc will return another floating point number which will be multiplied by the cardinalities of the two classes involved to compute the desired expected result size.

The difference between the function

```
my_procedure_1 (MYBOXES.description, "0,0,1,1"::box)
```

and the operator

```
MYBOXES.description === "0,0,1,1"::box
```

is that Postgres attempts to optimize operators and can decide to use an index to restrict the search space when operators are involved. However, there is no attempt to optimize functions,

and they are performed by brute force. Moreover, functions can have any number of arguments while operators are restricted to one or two.

Notes

Refer to the chapter on operators in the PostgreSQL User's Guide for further information.
Refer to DROP OPERATOR to delete user-defined operators from a database.

Usage

The following command defines a new operator, area-equality, for the BOX data type.

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    PROCEDURE = area_equal_procedure,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_procedure,
    HASHES,
    JOIN = area-join-procedure,
    SORT = <<<, <<<)
```

Compatibility

CREATE OPERATOR is a Postgres extension.

SQL92

There is no CREATE OPERATOR statement in SQL92.

CREATE RULE

Name

CREATE RULE Defines a new rule

```
CREATE RULE name
    AS ON event
    TO object [ WHERE condition ]
    DO [ INSTEAD ] [ action | NOTHING ]
```

Inputs

name

The name of a rule to create.

event

Event is one of select, update, delete or insert.

object

Object is either table or table.column.

condition

Any SQL WHERE clause. new or current can appear instead of an instance variable whenever an instance variable is permissible in SQL.

action

Any SQL statement. new or current can appear instead of an instance variable whenever an instance variable is permissible in SQL.

Outputs

CREATE

Message returned if the rule is successfully created.

Description

The semantics of a rule is that at the time an individual instance is accessed, updated, inserted or deleted, there is a current instance (for retrieves, updates and deletes) and a new instance (for updates and appends). If the event specified in the ON clause and the condition specified in the WHERE clause are true for the current instance, the action part of the rule is executed. First, however, values from fields in the current instance and/or the new instance are substituted for current.attribute-name and new.attribute-name.

The action part of the rule executes with the same command and transaction identifier as the user command that caused activation.

Notes

A caution about SQL rules is in order. If the same class name or instance variable appears in the event, the condition and the action parts of a rule, they are all considered different tuple variables. More accurately, new and current are the only tuple variables that are shared between these clauses. For example, the following two rules have the same semantics:

```
on update to EMP.salary where EMP.name = "Joe"
    do update EMP ( ... ) where ...
```

```
on update to EMP-1.salary where EMP-2.name = "Joe"
    do update EMP-3 ( ... ) where ...
```

Each rule can have the optional tag INSTEAD. Without this tag, action will be performed in addition to the user command when the event in the condition part of the rule occurs. Alternately, the action part will be done instead of the user command. In this later case, the action can be the keyword NOTHING.

When choosing between the rewrite and instance rule systems for a particular rule application, remember that in the rewrite system, current refers to a relation and some qualifiers whereas in the instance system it refers to an instance (tuple).

It is very important to note that the rewrite rule system will neither detect nor process circular rules. For example, though each of the following two rule definitions are accepted by Postgres, the retrieve command will cause Postgres to crash:

Example 16-1. Example of a circular rewrite rule combination.

```
create rule bad_rule_combination_1 is
    on select to EMP
    do instead select to TOYEMP

create rule bad_rule_combination_2 is
    on select to TOYEMP
    do instead select to EMP
```

This attempt to retrieve from EMP will cause Postgres to crash.

```
select * from EMP
```

You must have rule definition access to a class in order to define a rule on it. Use GRANT and REVOKE to change permissions.

Usage

Make Sam get the same salary adjustment as Joe:

```
create rule example_1 is
    on update EMP.salary where current.name = "Joe"
    do update EMP (salary = new.salary)
    where EMP.name = "Sam"
```

At the time Joe receives a salary adjustment, the event will become true and Joe's current instance and proposed new instance are available to the execution routines. Hence, his new salary is substituted into the action part of the rule which is subsequently executed. This propagates Joe's salary on to Sam.

Make Bill get Joe's salary when it is accessed:

```
create rule example_2 is
    on select to EMP.salary
    where current.name = "Bill"
    do instead
    select (EMP.salary) from EMP
        where EMP.name = "Joe"
```

Deny Joe access to the salary of employees in the shoe department (current_user returns the name of the current user):

```
create rule example_3 is
    on select to EMP.salary
    where current.dept = "shoe" and current_user = "Joe"
    do instead nothing
```

Create a view of the employees working in the toy department.

```
create TOYEMP(name = char16, salary = int4)

create rule example_4 is
    on select to TOYEMP
    do instead
    select (EMP.name, EMP.salary) from EMP
        where EMP.dept = "toy"
```

All new employees must make 5,000 or less

```
create rule example_5 is
    on insert to EMP where new.salary > 5000
    do update newset salary = 5000
```

Bugs

The object in a SQL rule cannot be an array reference and cannot have parameters.

Aside from the "oid" field, system attributes cannot be referenced anywhere in a rule. Among other things, this means that functions of instances (e.g., "foo(emp)" where "emp" is a class) cannot be called anywhere in a rule.

The rule system stores the rule text and query plans as text attributes. This implies that creation of rules may fail if the rule plus its various internal representations exceed some value that is on the order of one page (8KB).

Compatibility

CREATE RULE statement is a Postgres language extension.

SQL92

There is no CREATE RULE statement in SQL92.

CREATE SEQUENCE

Name

CREATE SEQUENCE Creates a new sequence number generator

```
CREATE SEQUENCE seqname
    [ INCREMENT increment ]
    [ MINVALUE  minvalue ]
    [ MAXVALUE  maxvalue ]
    [ START     start ]
    [ CACHE     cache ]
    [ CYCLE ]
```

Inputs

seqname

The name of a sequence to be created.

increment

The INCREMENT increment clause is optional. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is one (1).

minvalue

The optional clause MINVALUE minvalue determines the minimum value a sequence can generate. The defaults are 1 and -2147483647 for ascending and descending sequences, respectively.

maxvalue

Use the optional clause MAXVALUE maxvalue to determine the maximum value for the sequence. The defaults are 2147483647 and -1 for ascending and descending sequences, respectively.

start

The optional START start clause enables the sequence to begin anywhere. The default starting value is minvalue for ascending sequences and maxvalue for descending ones.

cache

The CACHE cache option enables sequence numbers to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e. no cache) and this is also the default.

CYCLE

The optional CYCLE keyword may be used to enable the sequence to continue when the maxvalue or minvalue has been reached by an ascending or descending sequence respectively.

```
   If the limit is reached, the next number generated will be
   whatever the minvalue or maxvalue is, as appropriate.
```

Outputs

```
 CREATE
    Message returned if the command is successful.
ERROR: amcreate: 'seqname' relation already exists
    If the sequence specified already exists.
ERROR: DefineSequence: START value (start) can't be > MAXVALUE
(maxvalue)
    If the specified starting value is out of range.
ERROR: DefineSequence: START value (start) can't be < MINVALUE
(minvalue)
    If the specified starting value is out of range.
ERROR: DefineSequence: MINVALUE (minvalue) can't be >= MAXVALUE
(maxvalue)
    If the minimum and maximum values are inconsistant.
```

Description

CREATE SEQUENCE will enter a new sequence number generator into the current data base. This involves creating and initialising a new single-row table with the name seqname. The generator will be "owned" by the user issuing the command.

After a sequence is created, you may use the function nextval(seqname) to get a new number from the sequence. The function currval('seqname') may be used to determine the number returned by the last call to nextval(seqname) for the specified sequence in the current session. The function setval('seqname', newvalue) may be used to set the current value of the specified sequence. The next call to nextval(seqname) will return the given value plus the sequence increment.

Use a query like

```
SELECT * FROM sequence_name;
```

to get the parameters of a sequence. Aside from fetching the original parameters, you can use

```
SELECT last_value FROM sequence_name;
```

to obtain the last value allocated by any backend. parameters, you can use

Low-level locking is used to enable multiple simultaneous calls to a generator.

Notes

Refer to the DROP SEQUENCE statement to remove a sequence.

Each backend uses its own cache to store allocated numbers. Numbers that are cached but not used in the current session will be lost, resulting in "holes" in the sequence.

Caution

Unexpected results may be obtained if a cache setting greater than one is used for a sequence object that will be used concurrently by multiple backends. Each backend will allocate "cache" successive sequence values during one access to the sequence object and increase the sequence object's last_value accordingly. Then, the next cache-1 uses of nextval within that backend simply return the preallocated values without touching the shared object. So, numbers allocated but not used in the current session will be lost. Furthermore, although multiple backends are guaranteed to allocate distinct sequence values, the values may be generated out of sequence when all the backends are considered. (For example, with a cache setting of 10, backend A might reserve values 1..10 and return nextval=1, then backend B might reserve values 11..20 and return nextval=11 before backend A has generated nextval=2.) Thus, with a cache setting of one it is safe to assume that nextval values are generated sequentially; with a cache setting greater than one you should only assume that the nextval values are all distinct, not that they are generated purely sequentially. Also, last_value will reflect the latest value reserved by any backend, whether or not it has yet been returned by nextval.

Usage

Create an ascending sequence called serial, starting at 101:

```
CREATE SEQUENCE serial START 101;
```

Select the next number from this sequence

```
SELECT NEXTVAL ('serial');

nextval
-------
    114
```

Use this sequence in an INSERT:

```
INSERT INTO distributors VALUES (NEXTVAL('serial'),'nothing');
```

Set the sequence value after a COPY FROM:

```
CREATE FUNCTION distributors_id_max() RETURNS INT4
       AS 'SELECT max(id) FROM distributors'
       LANGUAGE 'sql';
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', distributors_id_max());
END;
```

Compatibility

CREATE SEQUENCE is a Postgres language extension.

SQL92

There is no CREATE SEQUENCE statement in SQL92.

CREATE TABLE

Name

CREATE TABLE Creates a new table

```
CREATE TABLE table (
    column type
    [ DEFAULT value]
    [, NOT NULL ] [ ,UNIQUE ]
    [column_constraint_clause | PRIMARY KEY } [ ... ] ]
    [, ... ]
    [, PRIMARY KEY ( column [, ...] ) ]
    [, CHECK ( condition ) ]
    [, table_constraint_clause ]
    ) [ INHERITS ( inherited_table [, ...] ) ]
```

Inputs

table

The name of a new table to be created.

column

The name of a column.

type

The type of the column. This may include array specifiers. Refer to the PostgreSQL User's Guide for further information about data types and arrays.

DEFAULT value

A default value for a column. See the DEFAULT clause for more information.

column_constraint_clause

The optional column constraint clauses specify a list of integrity constraints or tests which new or updated entries must satisfy for an insert or update operation to succeed. Each constraint must evaluate to a boolean expression. Although SQL92 requires the column_constraint_clause to refer to that column only, Postgres allows multiple columns to be referenced within a single column constraint. See the column constraint clause for more information.

table_constraint_clause

The optional table CONSTRAINT clause specifies a list of integrity constraints which new or updated entries must satisfy for an insert or update operation to succeed. Each constraint must evaluate to a boolean expression. Multiple columns may be referenced within a single constraint. Only one PRIMARY KEY clause may be specified for a table; PRIMARY KEY column (a table constraint) and PRIMARY KEY (a column constraint) are mutually exclusive.. See the table constraint clause for more information.

INHERITS inherited_table

The optional INHERITS clause specifies a collection of table names from which this table automatically inherits all fields. If any inherited field name appears more than once, Postgres reports an error. Postgres automatically allows the

```
created table to inherit functions on tables above it in the
inheritance hierarchy.
```

Aside: Inheritance of functions is done according to the conventions of the Common Lisp Object System (CLOS).

Outputs

CREATE

```
Message returned if table is successfully created.
```

ERROR

```
Message returned if table creation failed. This is usually
accompanied by some descriptive text, such as:
amcreate: "table" relation already exists

which occurs at runtime, if the table specified already
exists in the database.
```

ERROR: DEFAULT: type mismatched

```
if data type of default value doesn't match the column
definition's data type.
```

Description

CREATE TABLE will enter a new table into the current data base. The table will be "owned" by the user issuing the command.

The new table is created as a heap with no initial data. A table can have no more than 1600 columns (realistically, this is limited by the fact that tuple sizes must be less than 8192 bytes), but this limit may be configured lower at some sites. A table cannot have the same name as a system catalog table.

DEFAULT Clause

```
DEFAULT value
```

Inputs

value

The possible values for the default value expression are:

a literal value

a user function

a niladic function

Outputs

Description

The DEFAULT clause assigns a default data value to a column (via a column definition in the CREATE TABLE statement). The data type of a default value must match the column definition's data type.

An INSERT operation that includes a column without a specified default value will assign the NULL value to the column if no explicit data value is provided for it. Default literal means that

the default is the specified constant value. Default niladic-function or user-function means that the default is the value of the specified function at the time of the INSERT.

There are two types of niladic functions:

niladic USER

CURRENT_USER / USER

See CURRENT_USER function

SESSION_USER

not yet supported

SYSTEM_USER

not yet supported

niladic datetime

CURRENT_DATE

See CURRENT_DATE function

CURRENT_TIME

See CURRENT_TIME function

CURRENT_TIMESTAMP

See CURRENT_TIMESTAMP function

In the current release (v6.4), Postgres evaluates all default expressions at the time the table is defined. Hence, functions which are "non-cacheable" such as CURRENT_TIMESTAMP may not produce the desired effect. For the particular case of date/time types, one can work around this behavior by using DEFAULT TEXT 'now' instead of DEFAULT 'now' or DEFAULT CURRENT_TIMESTAMP . This forces Postgres to consider the constant a string type and then to convert the value to timestamp at runtime.

Usage

To assign a constant value as the default for the columns did and number, and a string literal to the column did:

```
CREATE TABLE video_sales (
    did      VARCHAR(40) DEFAULT 'luso films',
    number   INTEGER DEFAULT 0,
    total    CASH DEFAULT '$0.0'
);
```

To assign an existing sequence as the default for the column did, and a literal to the column name:

```
CREATE TABLE distributors (
    did      DECIMAL(3)  DEFAULT NEXTVAL('serial'),
    name     VARCHAR(40) DEFAULT 'luso films'
);
```

Column CONSTRAINT Clause

```
[ CONSTRAINT name ] { NOT NULL | UNIQUE | PRIMARY KEY | CHECK
constraint } [, ...]
```

Inputs

name

An arbitrary name given to the integrity constraint. If name is not specified, it is generated from the table and column names, which should ensure uniqueness for name.

NOT NULL

The column is not allowed to contain NULL values. This is equivalent to the column constraint CHECK (column NOT NULL).

UNIQUE

The column must have unique values. In Postgres this is enforced by an implicit creation of a unique index on the table.

PRIMARY KEY

This column is a primary key, which implies that uniqueness is enforced by the system and that other tables may rely on this column as a unique identifier for rows. See PRIMARY KEY for more information.

constraint

The definition of the constraint.

Description

A Constraint is a named rule: an SQL object which helps define valid sets of values by putting limits on the results of INSERT, UPDATE or DELETE operations performed on a Base Table.

There are two ways to define integrity constraints: table constraints, covered later, and column constraints, covered here.

A column constraint is an integrity constraint defined as part of a column definition, and logically becomes a table constraint as soon as it is created. The column constraints available are:

PRIMARY KEY
REFERENCES
UNIQUE
CHECK
NOT NULL

Note: Postgres does not yet (at release 6.4) support REFERENCES integrity constraints. The parser accepts the REFERENCES syntax but ignores the clause.

NOT NULL Constraint

```
[ CONSTRAINT name ] NOT NULL
```

The NOT NULL constraint specifies a rule that a column may contain only non-null values.

The NOT NULL constraint is a column constraint only, and not allowed as a table constraint.

Outputs

status

ERROR: ExecAppend: Fail to add null value in not null attribute "column".

This error occurs at runtime if one tries to insert a null value into a column which has a NOT NULL constraint.

Description

Usage

Define two NOT NULL column constraints on the table distributors, one of which being a named constraint:

```
CREATE TABLE distributors (
    did      DECIMAL(3) CONSTRAINT no_null NOT NULL,
    name     VARCHAR(40) NOT NULL
);
```

UNIQUE Constraint

```
[ CONSTRAINT name ] UNIQUE
```

Inputs

CONSTRAINT name

An arbitrary label given to a constraint.

Outputs

status

ERROR: Cannot insert a duplicate key into a unique index.

This error occurs at runtime if one tries to insert a duplicate value into a column.

Description

The UNIQUE constraint specifies a rule that a group of one or more distinct columns of a table may contain only unique values.

The column definitions of the specified columns do not have to include a NOT NULL constraint to be included in a UNIQUE constraint. Having more than one null value in a column without a NOT NULL constraint, does not violate a UNIQUE constraint. (This deviates

from the SQL92 definition, but is a more sensible convention. See the section on compatibility for more details.).

Each UNIQUE column constraint must name a column that is different from the set of columns named by any other UNIQUE or PRIMARY KEY constraint defined for the table.

> Note: Postgres automatically creates a unique index for each UNIQUE constraint, to assure data integrity. See CREATE INDEX for more information.

Usage

Defines a UNIQUE column constraint for the table distributors. UNIQUE column constraints can only be defined on one column of the table:

```
CREATE TABLE distributors (
     did      DECIMAL(3),
     name     VARCHAR(40) UNIQUE
     );
```

which is equivalent to the following specified as a table constraint:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40),
    UNIQUE(name)
);
```

The CHECK Constraint

```
[ CONSTRAINT name ] CHECK ( condition [, ...] )
```

Inputs

name

An arbitrary name given to a constraint.

condition

Any valid conditional expression evaluating to a boolean result.

Outputs

status

ERROR: ExecAppend: rejected due to CHECK constraint "table_column".

This error occurs at runtime if one tries to insert an illegal value into a column subject to a CHECK constraint.

Description

The CHECK constraint specifies a restriction on allowed values within a column. The CHECK constraint is also allowed as a table constraint.

The SQL92 CHECK column constraints can only be defined on, and refer to, one column of the table. Postgres does not have this restriction.

PRIMARY KEY Constraint

```
[ CONSTRAINT name ] PRIMARY KEY
```

Inputs

CONSTRAINT name

An arbitrary name for the constraint.

Outputs

ERROR: Cannot insert a duplicate key into a unique index.

This occurs at run-time if one tries to insert a duplicate value into a column subject to a PRIMARY KEY constraint.

Description

The PRIMARY KEY column constraint specifies that a column of a table may contain only unique (non-duplicate), non-NULL values. The definition of the specified column does not have to include an explicit NOT NULL constraint to be included in a PRIMARY KEY constraint.

Only one PRIMARY KEY can be specified for a table.

Notes

Postgres automatically creates a unique index to assure data integrity. (See CREATE INDEX statement)

The PRIMARY KEY constraint should name a set of columns that is different from other sets of columns named by any UNIQUE constraint defined for the same table, since it will result in duplication of equivalent indexes and unproductive additional runtime overhead. However, Postgres does not specifically disallow this.

Table CONSTRAINT Clause

```
[ CONSTRAINT name ] { PRIMARY KEY |  UNIQUE } ( column [, ...] )
[ CONSTRAINT name ] CHECK ( constraint )
```

Inputs

CONSTRAINT name

An arbitrary name given to an integrity constraint.

column [, ...]

The column name(s) for which to define a unique index and, for PRIMARY KEY, a NOT NULL constraint.

CHECK (constraint)

A boolean expression to be evaluated as the constraint.

Outputs

The possible outputs for the table constraint clause are the same as for the corresponding portions of the column constraint clause.

Description

A table constraint is an integrity constraint defined on one or more columns of a base table. The four variations of "Table Constraint" are:

UNIQUE
CHECK
PRIMARY KEY
FOREIGN KEY

Note: Postgres does not yet (as of version 6.4) support FOREIGN KEY integrity constraints. The parser understands the FOREIGN KEY syntax, but only prints a notice and otherwise ignores the clause. Foreign keys may be partially emulated by triggers (See the CREATE TRIGGER statement).

UNIQUE Constraint

```
[ CONSTRAINT name ] UNIQUE ( column [, ...] )
```

Inputs

CONSTRAINT name

An arbitrary name given to a constraint.

column

A name of a column in a table.

Outputs

status

ERROR: Cannot insert a duplicate key into a unique index.

This error occurs at runtime if one tries to insert a duplicate value into a column.

Description

The UNIQUE constraint specifies a rule that a group of one or more distinct columns of a table may contain only unique values. The behavior of the UNIQUE table constraint is the same as that for column constraints, with the additional capability to span multiple columns.

See the section on the UNIQUE column constraint for more details.

Usage

Define a UNIQUE table constraint for the table distributors:

```
CREATE TABLE distributors (
    did      DECIMAL(03),
    name     VARCHAR(40),
    UNIQUE(name)
);
```

PRIMARY KEY Constraint

```
    [ CONSTRAINT name ] PRIMARY KEY ( column [, ...] )
```

Inputs

CONSTRAINT name

An arbitrary name for the constraint.

column [, ...]

The names of one or more columns in the table.

Outputs

status

ERROR: Cannot insert a duplicate key into a unique index.

This occurs at run-time if one tries to insert a duplicate value into a column subject to a PRIMARY KEY constraint.

Description

The PRIMARY KEY constraint specifies a rule that a group of one or more distinct columns of a table may contain only unique, (non duplicate), non-null values. The column definitions of the specified columns do not have to include a NOT NULL constraint to be included in a PRIMARY KEY constraint. The PRIMARY KEY table constraint is similar to that for column constraints, with the additional capability of encompassing multiple columns.

Refer to the section on the PRIMARY KEY column constraint for more information.

Usage

Create table films and table distributors

```
CREATE TABLE films (
       code      CHARACTER(5) CONSTRAINT firstkey PRIMARY KEY,
       title     CHARACTER VARYING(40) NOT NULL,
       did       DECIMAL(3) NOT NULL,
       date_prod DATE,
       kind      CHAR(10),
       len       INTERVAL HOUR TO MINUTE
       );


CREATE TABLE distributors (
       did      DECIMAL(03) PRIMARY KEY DEFAULT NEXTVAL('serial'),
       name     VARCHAR(40) NOT NULL CHECK (name <> '')
       );
```

Create a table with a 2-dimensional array

```
CREATE TABLE array (
       vector INT[][]
       );
```

Define a UNIQUE table constraint for the table films. UNIQUE table constraints can be defined on one or more columns of the table

```
CREATE TABLE films (
    code      CHAR(5),
    title     VARCHAR(40),
    did       DECIMAL(03),
    date_prod DATE,
    kind      CHAR(10),
    len       INTERVAL HOUR TO MINUTE,
    CONSTRAINT production UNIQUE(date_prod)
    );
```

Define a CHECK column constraint.

```
CREATE TABLE distributors (
     did      DECIMAL(3) CHECK (did > 100),
     name     VARCHAR(40)
     );
```

Define a CHECK table constraint

```
CREATE TABLE distributors (
     did      DECIMAL(3),
     name     VARCHAR(40)
     CONSTRAINT con1 CHECK (did > 100 AND name > '')
     );
```

Define a PRIMARY KEY table constraint for the table films. PRIMARY KEY table constraints can be defined on one or more columns of the table

```
CREATE TABLE films (
    code      CHAR(05),
    title     VARCHAR(40),
    did       DECIMAL(03),
    date_prod DATE,
    kind      CHAR(10),
    len       INTERVAL HOUR TO MINUTE,
    CONSTRAINT code_title PRIMARY KEY(code,title)
    );
```

Defines a PRIMARY KEY column constraint for table distributors. PRIMARY KEY column constraints can only be defined on one column of the table (the following two examples are equivalent)

```
CREATE TABLE distributors (
     did      DECIMAL(03),
     name     CHAR VARYING(40),
     PRIMARY KEY(did)
     );

CREATE TABLE distributors (
     did      DECIMAL(03) PRIMARY KEY,
     name     VARCHAR(40)
     );
```

Notes

CREATE TABLE/INHERITS is a Postgres language extension.

Compatibility

SQL92

In addition to the normal CREATE TABLE, SQL92 also defines a CREATE TEMPORARY TABLE statement:

```
   CREATE [ {GLOBAL | LOCAL} ] TEMPORARY TABLE table (
        column type [DEFAULT value] [CONSTRAINT column_constraint] [,
...] )
        [CONSTRAINT table_constraint ]
        [ ON COMMIT {DELETE | PRESERVE} ROWS ]
```

For temporary tables, the CREATE TEMPORARY TABLE statement names a new table and defines the table's columns and constraints.

The optional ON COMMIT clause of CREATE TEMPORARY TABLE specifies whether or not the temporary table should be emptied of rows whenever COMMIT is executed. If the ON COMMIT clause is omitted, the default option, ON COMMIT DELETE ROWS, is assumed.

To create a temporary table:

```
CREATE TEMPORARY TABLE actors (
       id      DECIMAL(03),
       name    VARCHAR(40),
       CONSTRAINT actor_id CHECK (id < 150)
       ) ON COMMIT DELETE ROWS
```

Temporary tables are not currently available in Postgres.

> Tip: In the current release of Postgres (v6.4), to create a temporary table you must create and drop the table by explicit commands.

UNIQUE clause

SQL92 specifies some additional capabilities for UNIQUE:

Table Constraint definition

```
[ CONSTRAINT name ]
UNIQUE ( column [, ...] )
[ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
[ [ NOT ] DEFERRABLE ]
```

Column Constraint definition

```
[ CONSTRAINT name ]
UNIQUE
[ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
[ [ NOT ] DEFERRABLE ]
```

NOT NULL clause

SQL92 specifies some additional capabilities for NOT NULL:

```
[ CONSTRAINT name ] NOT NULL
[ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
[ [ NOT ] DEFERRABLE ]
```

CONSTRAINT clause

SQL92 specifies some additional capabilities for constraints, and also defines assertions and domain constraints.

Note: Postgres does not yet support either domains or assertions.

An assertion is a special type of integrity constraint and share the same namespace as other constraints. However, an assertion is not necessarily dependent on one particular base table as constraints are, so SQL-92 provides the CREATE ASSERTION statement as an alternate method for defining a constraint:

```
CREATE ASSERTION name CHECK ( condition )
```

Domain constraints are defined by CREATE DOMAIN or ALTER DOMAIN statements:

Domain constraint:

```
[ CONSTRAINT name ]
    CHECK constraint
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

Table constraint definition:

```
[ CONSTRAINT name ]
  { PRIMARY KEY constraint |
    FOREIGN KEY constraint |
    UNIQUE constraint |
    CHECK constraint }
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

Column constraint definition:

```
[ CONSTRAINT name ]
   { NOT NULL constraint |
     PRIMARY KEY constraint |
     FOREIGN KEY constraint |
     UNIQUE constraint |
     CHECK constraint }
     [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
     [ [ NOT ] DEFERRABLE ]
```

A CONSTRAINT definition may contain one deferment attribute clause and/or one initial constraint mode clause, in any order.

NOT DEFERRABLE

means that the Constraint must be checked for violation of its rule after the execution of every SQL statement.

DEFERRABLE

means that checking of the Constraint may be deferred until some later time, but no later than the end of the current transaction.

The constraint mode for every Constraint always has an initial default value which is set for that Constraint at the beginning of a transaction.

INITIALLY IMMEDIATE

means that, as of the start of the transaction, the Constraint must be checked for violation of its rule after the execution of every SQL statement.

INITIALLY DEFERRED

means that, as of the start of the transaction, checking of the Constraint may be deferred until some later time, but no later than the end of the current transaction.

CHECK clause

SQL92 specifies some additional capabilities for CHECK in either table or column constraints.

table constraint definition:

```
[ CONSTRAINT name ]
 CHECK ( VALUE condition )
 [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
 [ [ NOT ] DEFERRABLE ]
```

column constraint definition:

```
 [ CONSTRAINT name ]
 CHECK ( VALUE condition )
 [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
 [ [ NOT ] DEFERRABLE ]
```

PRIMARY KEY clause

SQL92 specifies some additional capabilities for PRIMARY KEY:

Table Constraint definition:

```
 [ CONSTRAINT name ]
 PRIMARY KEY ( column [, ...] )
 [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
 [ [ NOT ] DEFERRABLE ]
```

Column Constraint definition:

```
 [ CONSTRAINT name ]
 PRIMARY KEY
 [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
 [ [ NOT ] DEFERRABLE ]
```

CREATE TABLE AS

Name

CREATE TABLE AS Creates a new table

```
CREATE TABLE table [ ( column [, ...] ) ] AS select_clause
```

Inputs

table

The name of a new table to be created.

column

The name of a column. Multiple column names can be specified using a comma-delimited list of column names.

select_clause

A valid query statement. Refer to SELECT for a description of the allowed syntax.

Outputs

Refer to CREATE TABLE and SELECT for a summary of possible output messages.

Description

CREATE TABLE AS enables a table to be created from the contents of an existing table. It has functionality equivalent to SELECT TABLE INTO, but with perhaps a more obvious syntax.

CREATE TRIGGER

Name

CREATE TRIGGER Creates a new trigger

```
CREATE TRIGGER name { BEFORE | AFTER }
    { event [OR ...] }
    ON table FOR EACH { ROW | STATEMENT }
    EXECUTE PROCEDURE funcname ( arguments )
```

Inputs

name
: The name of an existing trigger.

table
: The name of a table.

event
: One of INSERT, DELETE or UPDATE.

funcname
: A user-supplied function.

Outputs

CREATE
: This message is returned if the trigger is successfully created.

Description

CREATE TRIGGER will enter a new trigger into the current data base. The trigger will be associated with the relation relname and will execute the specified function funcname.

The trigger can be specified to fire either before the operation is attempted on a tuple (before constraints are checked and the INSERT, UPDATE or DELETE is attempted) or after the operation has been attempted (e.g. after constraints are checked and the INSERT, UPDATE or DELETE has completed). If the trigger fires before the event, the trigger may skip the operation for the current tuple, or change the tuple being inserted (for INSERT and UPDATE

operations only). If the trigger fires after the event, all changes, including the last insertion, update, or deletion, are "visible" to the trigger.

Refer to the chapters on SPI and Triggers in the PostgreSQL Programmer's Guide for more information.

Notes

CREATE TRIGGER is a Postgres language extension.

Only the relation owner may create a trigger on this relation.

As of the current release (v6.4), STATEMENT triggers are not implemented.

Refer to DROP TRIGGER for information on how to remove triggers.

Usage

Check if the specified distributor code exists in the distributors table before appending or updating a row in the table films:

```
CREATE TRIGGER if_dist_exists
    BEFORE INSERT OR UPDATE ON films FOR EACH ROW
    EXECUTE PROCEDURE check_primary_key ('did', 'distributors', 'did');
```

Before cancelling a distributor or updating its code, remove every reference to the table films:

```
CREATE TRIGGER if_film_exists
    BEFORE DELETE OR UPDATE ON distributors FOR EACH ROW
    EXECUTE PROCEDURE check_foreign_key (1, 'CASCADE', 'did', 'films',
'did');
```

Compatibility

SQL92

There is no CREATE TRIGGER in SQL92.

The second example above may also be done by using a FOREIGN KEY constraint as in:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40),
    CONSTRAINT if_film_exists
        FOREIGN KEY(did) REFERENCES films
            ON UPDATE CASCADE ON DELETE CASCADE
);
```

However, foreign keys are not yet implemented (as of version 6.4) in Postgres.

CREATE TYPE

Name

CREATE TYPE Defines a new base data type

```
CREATE TYPE typename (
        INPUT          = input_function
      , OUTPUT         = output_function
      , INTERNALLENGTH = (internallength | VARIABLE)
    [ , EXTERNALLENGTH = (externallength | VARIABLE) ]
    [ , ELEMENT        = element ]
    [ , DELIMITER      = delimiter ]
    [ , DEFAULT        = "default" ]
    [ , SEND           = send_function ]
    [ , RECEIVE        = receive_function ]
    [ , PASSEDBYVALUE ]
 )
```

Inputs

typename

The name of a type to be created.

INTERNALLENGTH internallength

A literal value, which specifies the internal length of the new type.

EXTERNALLENGTH externallength

A literal value, which specifies the external length of the new type.

INPUT input_function

The name of a function, created by CREATE FUNCTION, which converts data from its external form to the type’s internal form.

OUTPUT output_function

The name of a function, created by CREATE FUNCTION, which converts data from its internal form to a form suitable for display.

element

The type being created is an array; this specifies the type of the array elements.

delimiter

The delimiter character for the array.

default

The default text to be displayed to indicate "data not present"

send_function

The name of a function, created by CREATE FUNCTION, which converts data of this type into a form suitable for transmission to another machine.

receive_function

The name of a function, created by CREATE FUNCTION, which converts data of this type from a form suitable for transmission from another machine to internal form.

Outputs

CREATE
    Message returned if the type is successfully created.

Description

CREATE TYPE allows the user to register a new user data type with Postgres for use in the current data base. The user who defines a type becomes its owner. Typename is the name of the new type and must be unique within the types defined for this database.

CREATE TYPE requires the registration of two functions (using create function) before defining the type. The representation of a new base type is determined by input_function, which converts the type's external representation to an internal representation usable by the operators and functions defined for the type. Naturally, output_function performs the reverse transformation. Both the input and output functions must be declared to take one or two arguments of type "opaque".

New base data types can be fixed length, in which case internallength is a positive integer, or variable length, in which case Postgres assumes that the new type has the same format as the Postgres-supplied data type, "text". To indicate that a type is variable-length, set internallength to VARIABLE. The external representation is similarly specified using the externallength keyword.

To indicate that a type is an array and to indicate that a type has array elements, indicate the type of the array element using the element keyword. For example, to define an array of 4 byte integers ("int4"), specify

```
ELEMENT = int4
```

To indicate the delimiter to be used on arrays of this type, delimiter can be set to a specific character. The default delimiter is the comma (",").

A default value is optionally available in case a user wants some specific bit pattern to mean "data not present." Specify the default with the DEFAULT keyword.

The optional functions send_function and receive_function are used when the application program requesting Postgres services resides on a different machine. In this case, the machine on which Postgres runs may use a format for the data type different from that used on the remote machine. In this case it is appropriate to convert data items to a standard form when sending from the server to the client and converting from the standard format to the machine specific format when the server receives the data from the client. If these functions are not specified, then it is assumed that the internal format of the type is acceptable on all relevant

machine architectures. For example, single characters do not have to be converted if passed from a Sun-4 to a DECstation, but many other types do.

The optional flag, PASSEDBYVALUE, indicates that operators and functions which use this data type should be passed an argument by value rather than by reference. Note that you may not pass by value types whose internal representation is more than four bytes.

For new base types, a user can define operators, functions and aggregates using the appropriate facilities described in this section.

Array Types

Two generalized built-in functions, array_in and array_out, exist for quick creation of variable-length array types. These functions operate on arrays of any existing Postgres type.

Large Object Types

A "regular" Postgres type can only be 8192 bytes in length. If you need a larger type you must create a Large Object type. The interface for these types is discussed at length in Section 7, the large object interface. The length of all large object types is always VARIABLE.

Examples

This command creates the box data type and then uses the type in a class definition:

```
CREATE TYPE box (INTERNALLENGTH = 8,
     INPUT = my_procedure_1, OUTPUT = my_procedure_2)

CREATE TABLE myboxes (id INT4, description box)
```

This command creates a variable length array type with integer elements.

```
CREATE TYPE int4array
   (INPUT = array_in, OUTPUT = array_out,
    INTERNALLENGTH = VARIABLE, ELEMENT = int4)

CREATE TABLE myarrays (id int4, numbers int4array)
```

This command creates a large object type and uses it in a class definition.

```
CREATE TYPE bigobj
   (INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE)

CREATE TABLE big_objs (id int4, obj bigobj)
```

Restrictions

Type names cannot begin with the underscore character ("_") and can only be 15 characters long. This is because Postgres silently creates an array type for each base type with a name consisting of the base type's name prepended with an underscore.

Notes

Refer to DROP TYPE to remove an existing type.

See also CREATE FUNCTION, CREATE OPERATOR and the chapter on Large Objects in the PostgreSQL Programmer's Guide.

Compatibility

SQL3

CREATE TYPE is an SQL3 statement.

CREATE USER

Name

CREATE USER Creates account information for a new user

```
CREATE USER username
    [ WITH PASSWORD password ]
    [ CREATEDB   | NOCREATEDB ]
    [ CREATEUSER | NOCREATEUSER ]
    [ IN GROUP     groupname [, ...] ]
    [ VALID UNTIL  'abstime' ]
```

Inputs

username

The name of the user.

password

The WITH PASSWORD clause sets the user's password within the "pg_shadow" table. For this reason, "pg_shadow" is no longer accessible to the instance of Postgres that the Postgres user's password is initially set to NULL. When a user's password in the "pg_shadow" table is NULL, user authentication proceeds as it historically has (HBA, PG_PASSWORD, etc). However, if a password is set for a user, a new authentication system supplants any other configured for the Postgres instance, and the password stored in the "pg_shadow" table is used for authentication. For more details on how this authentication system functions see pg_crypt(3). If the WITH PASSWORD clause is omitted, the user's password is set to the empty string with equates to a NULL value in the authentication system mentioned above.

CREATEDB/NOCREATEDB

These clauses define a user's ability to create databases. If CREATEDB is specified, the user being defined will be allowed to create his own databases. Using NOCREATEDB will deny a user the ability to create databases. If this clause is omitted, NOCREATEDB is used by default.

CREATEUSER/NOCREATEUSER

These clauses determine whether a user will be permitted to create new users in an instance of Postgres. Omitting this clause will set the user's value of this attribute to be NOCREATEUSER.

groupname

A name of a group into which to insert the user as a new member.

abstime

The VALID UNTIL clause sets an absolute time after which the user's Postgres login is no longer valid. Please note that if a user does not have a password defined in the "pg_shadow" table, the valid until date will not be checked during user authentication. If this clause is omitted, a

```
NULL value is stored in "pg_shadow" for this attribute, and
the login will be valid for all time.
```

Outputs

```
CREATE USER
   Message returned if the command completes successfully.
ERROR: removeUser: user "username" does not exist
   if "username" not found.
```

Description

CREATE USER will add a new user to an instance of Postgres.

The new user will be given a usesysid of: 'SELECT MAX(usesysid) + 1 FROM pg_shadow'. This means that Postgres users' usesysids will not correspond to their operating system(OS) user ids. The exception to this rule is the 'postgres' user, whose OS user id is used as the usesysid during the initdb process. If you still want the OS user id and the usesysid to match for any given user, use the "createuser" script provided with the Postgres distribution.

Notes

CREATE USER statement is a Postgres language extension.

Use DROP USER or ALTER USER statements to remove or modify a user account.

Refer to the pg_shadow table for further information.

```
Table    = pg_shadow
+--------------------------+--------------------------+-------+
|          Field           |          Type            | Length|
+--------------------------+--------------------------+-------+
| usename                  | name                     |    32 |
| usesysid                 | int4                     |     4 |
| usecreatedb              | bool                     |     1 |
| usetrace                 | bool                     |     1 |
| usesuper                 | bool                     |     1 |
| usecatupd                | bool                     |     1 |
| passwd                   | text                     |   var |
| valuntil                 | abstime                  |     4 |
+--------------------------+--------------------------+-------+
```

Usage

Create a user with no password:

```
CREATE USER jonathan
```

Create a user with a password:

```
CREATE USER davide WITH PASSWORD jw8s0F4
```

Create a user with a password, whose account is valid until the end of 2001. Note that after one second has ticked in 2002, the account is not valid:

```
CREATE USER miriam WITH PASSWORD jw8s0F4 VALID UNTIL 'Jan 1 2002'
```

Create an account where the user can create databases:

```
CREATE USER manuel WITH PASSWORD jw8s0F4 CREATEDB
```

Compatibility

SQL92

There is no CREATE USER statement in SQL92.

CREATE VIEW

Name

CREATE VIEW Constructs a virtual table

```
CREATE VIEW view
    AS SELECT query
```

Inputs

`view`

The name of a view to be created.

`query`

An SQL query which will provide the columns and rows of the view.

Refer to the SELECT statement for more information about valid arguments.

Outputs

`CREATE`

The message returned if the view is successfully created.

`WARN amcreate: "view" relation already exists`

This error occurs if the view specified already exists in the database.

`NOTICE create: attribute named "column" has an unknown type`

The view will be created having a column with an unknown type if you do not specify it. For example, the following command gives an error:

```
CREATE VIEW vista AS SELECT 'Hello World'
```

whereas this command does not:

```
CREATE VIEW vista AS SELECT 'Hello World'::text
```

Description

CREATE VIEW will define a view of a table. This view is not physically materialized. Specifically, a query rewrite retrieve rule is automatically generated to support retrieve operations on views.

Notes

Use the DROP VIEW statement to drop views.

Bugs

Currently, views are read only.

Usage

Create a view consisting of all Comedy films:

```
CREATE VIEW kinds AS
```

```
    SELECT *
    FROM films
    WHERE kind = 'Comedy';

SELECT * FROM kinds;

code |title                     |did| date_prod|kind      |len
-----+--------------------------+---+----------+----------+------
UA502|Bananas                   |105|1971-07-13|Comedy    | 01:22
C_701|There's a Girl in my Soup|107|1970-06-11|Comedy    | 01:36
```

Compatibility

SQL92

SQL92 specifies some additional capabilities for the CREATE VIEW statement:

```
CREATE VIEW view [ column [, ...] ]
    AS SELECT expression [AS colname] [, ...]
    FROM table
    [ WHERE condition ]
    [ WITH [ CASCADE | LOCAL ] CHECK OPTION ]
```

CHECK OPTION

This option is to do with updatable views. All INSERTs and UPDATEs on the view will be checked to ensure data satisfy the view-defining condition. If they do not, the update will be rejected.

LOCAL

Check for integrity on this view.

CASCADE

Check for integrity on this view and on any dependent view. CASCADE is assumed if neither CASCADE nor LOCAL is specified.

DECLARE

Name

DECLARE Defines a cursor for table access

```
DECLARE cursor [ BINARY ] [ INSENSITIVE ] [ SCROLL ]
    CURSOR FOR query
    [ FOR { READ ONLY | UPDATE [ OF column [, ...] ] ]
```

Inputs

cursor

The name of the cursor to be used in subsequent FETCH operations..

BINARY

Causes the cursor to fetch data in binary rather than in text format.

INSENSITIVE

SQL92 keyword indicating that data retrieved from the cursor should be unaffected by updates from other processes or cursors. Since cursor operations occur within transactions in Postgres this is always the case. This keyword has no effect.

SCROLL

SQL92 keyword indicating that data may be retrieved in multiple rows per FETCH operation. Since this is allowed at all times by Postgres this keyword has no effect.

query

An SQL query which will provide the rows to be governed by the cursor. Refer to the SELECT statement for further information about valid arguments.

READ ONLY

SQL92 keyword indicating that the cursor will be used in a readonly mode. Since this is the only cursor access mode available in Postgres this keyword has no effect.

UPDATE

SQL92 keyword indicating that the cursor will be used to update tables. Since cursor updates are not currently

```
   supported in Postgres this keyword provokes an informational
   error message.
column
   Column(s) to be updated. Since cursor updates are not
   currently supported in Postgres the UPDATE clause provokes
   an informational error message.
```

Outputs

```
SELECT
    The message returned if the SELECT is run successfully.
NOTICE BlankPortalAssignName: portal "cursor" already exists
    This error occurs if cursor "cursor" is already declared.
ERROR: Named portals may only be used in begin/end transaction
blocks
   This error occurs if the cursor is not declared within a
   transaction block.
```

Description

DECLARE allows a user to create cursors, which can be used to retrieve a small number of rows at a time out of a larger query. Cursors can return data either in text or in binary foramt.

Normal cursors return data in text format, either ASCII or another encoding scheme depending on how the Postgres backend was built. Since data is stored natively in binary format, the system must do a conversion to produce the text format. In addition, text formats are often larger in size than the corresponding binary format. Once the information comes back in text form, the client application may have to convert it to a binary format to manipulate it anyway.

BINARY cursors give you back the data in the native binary representation. So binary cursors will tend to be a little faster since they suffer less conversion overhead.

As an example, if a query returns a value of one from an integer column, you would get a string of ’1’ with a default cursor whereas with a binary cursor you would get a 4-byte value equal to control-A (’^A’).

> **Caution**
>
> BINARY cursors should be used carefully. User applications such as psql are not aware of binary cursors and expect data to come back in a text format.

However, string representation is architecture-neutral whereas binary representation can differ between different machine architectures. Therefore, if your client machine and server machine

use different representations (e.g. "big-endian" versus "little-endian"), you will probably not want your data returned in binary format.

> Tip: If you intend to display the data in ASCII, getting it back in ASCII will save you some effort on the client side.

Notes

Cursors are only available in transactions.

Postgres does not have an explicit OPEN cursor statement; a cursor is considered to be open when it is declared.

> Note: In SQL92 cursors are only available in embedded applications. ecpg, the embedded SQL preprocessor for Postgres, supports the SQL92 conventions, including those involving DECLARE and OPEN statements.

Usage

To declare a cursor:

```
DECLARE liahona CURSOR
    FOR SELECT * FROM films;
```

Compatibility

SQL92

SQL92 allows cursors only in embedded SQL and in modules. Postgres permits cursors to be used interactively. SQL92 allows embedded or modular cursors to update database information. All Postgres cursors are readonly. The BINARY keyword is a Postgres extension.

DELETE

Name

DELETE Deletes rows from a table

```
DELETE FROM table [ WHERE condition ]
```

Inputs

table

The name of an existing table.

condition

This is an SQL selection query which returns the rows which are to be deleted.

Refer to the SELECT statement for further description of the WHERE clause.

Outputs

DELETE count

Message returned if items are successfully deleted. The count is the number of rows deleted.

If count is 0, no rows were deleted.

Description

DELETE removes rows which satisfy the WHERE condition, from the specified table.

If the condition is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.

You must have write access to the table in order to modify it, as well as read access to any table whose values are read in the condition.

Usage

Remove all films but musicals:

```
DELETE FROM films WHERE kind <> 'Musical';

SELECT * FROM films;

code |title                    |did| date_prod|kind      |len
-----+-------------------------+---+----------+----------+------
UA501|West Side Story          |105|1961-01-03|Musical   | 02:32
TC901|The King and I           |109|1956-08-11|Musical   | 02:13
WD101|Bed Knobs and Broomsticks|111|          |Musical   | 01:57
(3 rows)
```

Clear the table films:

```
DELETE FROM films;

SELECT * FROM films;
code|title|did|date_prod|kind|len
----+-----+---+---------+----+---
```

```
(0 rows)
```

Compatibility

SQL92

SQL92 allows a positioned DELETE statement:

```
DELETE FROM table WHERE CURRENT OF cursor
```

where cursor identifies an open cursor. Interactive cursors in Postgres are read-only.

DROP AGGREGATE

Name

DROP AGGREGATE Removes the definition of an aggregate function

```
DROP AGGREGATE name type
```

Inputs

name

The name of an existing aggregate function.

type

The type of an existing aggregate function. (Refer to the PostgreSQL User's Guide for further information about data types).

Outputs

DROP

Message returned if the command is successful.

WARN RemoveAggregate: aggregate 'name' for 'type' does not exist

This message occurs if the aggregate function specified does not exist in the database.

Description

DROP AGGREGATE will remove all references to an existing aggregate definition. To execute this command the current user must be the owner of the aggregate.

Notes

The DROP AGGREGATE statement is a Postgres language extension.

Refer to the CREATE AGGREGATE statement to create aggregate functions.

Usage

To remove the myavg aggregate for type int4:

```
DROP AGGREGATE myavg int4;
```

Compatibility

SQL92

There is no DROP AGGREGATE statement in SQL92.

DROP DATABASE

Name

DROP DATABASE Destroys an existing database

```
DROP DATABASE name
```

Inputs

`name`

The name of an existing database to remove.

Outputs

`DESTROYDB`

This message is returned if the command is successful.

`WARN: destroydb: database "name" does not exist.`

This message occurs if the specified database does not exist.

`ERROR: destroydb cannot be executed on an open database`

This message occurs if the specified database does not exist.

Description

DROP DATABASE removes the catalog entries for an existing database and deletes the directory containing the data. It can only be executed by the database administrator (See the CREATE DATABASE command for details).

Notes

DROP DATABASE statement is a Postgres language extension.

> Tip: This query cannot be executed while connected to the target database. It is usually preferable to use the destroydb script instead.

Refer to the CREATE DATABASE statement for information on how to create a database.

Compatibility

SQL92

There is no DROP DATABASE in SQL92.

DROP FUNCTION

Name

DROP FUNCTION Removes a user-defined C function

```
DROP FUNCTION name ( [ type [, ...] ] )
```

Inputs

`name`
The name of an existing function.

`type`
The type of function parameters.

Outputs

`DROP`
Message returned if the command completes successfully.

`WARN RemoveFunction: Function "name" ("types") does not exist`
This message is given if the function specified does not exist in the current database.

Description

DROP FUNCTION will remove references to an existing C function. To execute this command the user must be the owner of the function. The input argument types to the function must be specified, as only the function with the given name and argument types will be removed.

Notes

Refer to CREATE FUNCTION to create aggregate functions.

Usage

This command removes the square root function:

```
DROP FUNCTION sqrt(int4);
```

Bugs

No checks are made to ensure that types, operators or access methods that rely on the function have been removed first.

Compatibility

DROP FUNCTION is a Postgres language extension.

SQL/PSM

SQL/PSM is a proposed standard to enable function extensibility. The SQL/PSM DROP FUNCTION statement has the following syntax:

```
DROP [ SPECIFIC ] FUNCTION name { RESTRICT | CASCADE }
```

DROP INDEX

Name

DROP INDEX Removes an index from a database

```
DROP INDEX index_name
```

Inputs

`index_name`
    The name of the index to remove.

Outputs

`DROP`
    The message returned if the index is successfully dropped.

`ERROR: index "index_name" nonexistent`
    This message occurs if index_name is not an index in the database.

Description

DROP INDEX drops an existing index from the database system. To execute this command you must be the owner of the index.

Notes

DROP INDEX is a Postgres language extension.

Refer to the CREATE INDEX statement for information on how to create indexes.

Usage

This command will remove the title_idx index:

```
DROP INDEX title_idx;
```

Compatibility

SQL92

SQL92 defines commands by which to access a generic relational database. Indexes are an implementation-dependent feature and hence there are no index-specific commands or definitions in the SQL92 language.

DROP LANGUAGE

Name

DROP LANGUAGE Removes a user-defined procedural language

```
DROP PROCEDURAL LANGUAGE 'langname'
```

Inputs

`langname`

The name of an existing language.

Outputs

`DROP`

This message is returned if the language is successfully dropped.

`ERROR: Language "langname" doesn't exist`

This message occurs if the language "langname" is not found.

Description

DROP PROCEDURAL LANGUAGE will remove the definition of the previously registered procedural language having the name 'langname'.

Notes

The DROP PROCEDURAL LANGUAGE statement is a Postgres language extension.

Refer to CREATE PROCEDURAL LANGUAGE for information on how to create procedural languages.

Bugs

No checks are made if functions or trigger procedures registered in this language still exist. To re-enable them without having to drop and recreate all the functions, the pg_proc's prolang attribute of the functions must be adjusted to the new object ID of the recreated pg_language entry for the PL.

Usage

This command removes the PL/Sample language:

```
DROP PROCEDURAL LANGUAGE 'plsample'
```

Compatibility

SQL92

There is no DROP PROCEDURAL LANGUAGE in SQL92.

DROP OPERATOR

Name

DROP OPERATOR Removes an operator from the database

```
DROP OPERATOR id ( type | NONE [,...] )
```

Inputs

id

The identifier of an existing operator.

type

The type of function parameters.

Outputs

DROP

The message returned if the command is successful.

ERROR: RemoveOperator: binary operator 'id' taking 'type1' and 'type2' does not exist

This message occurs if the specified binary operator does not exist.

ERROR: RemoveOperator: left unary operator 'id' taking 'type' does not exist

This message occurs if the specified left unary operator specified does not exist.

ERROR: RemoveOperator: right unary operator 'id' taking 'type' does not exist

This message occurs if the specified right unary operator specified does not exist.

Description

The DROP OPERATOR statement drops an existing operator from the database. To execute this command you must be the owner of the operator.

The left or right type of a left or right unary operator, respectively, may be specified as NONE.

Notes

The DROP OPERATOR statement is a Postgres language extension.

Refer to CREATE OPERATOR for information on how to create operators.

It is the user's responsibility to remove any access methods and operator classes that rely on the deleted operator.

Usage

Remove power operator a^n for int4:

```
DROP OPERATOR ^ (int4, int4);
```

Remove left unary operator !a for booleans:

```
DROP OPERATOR ! (none, bool);
```

Remove right unary factorial operator a! for int4:

```
DROP OPERATOR ! (int4, none);
```

Compatibility

SQL92

There is no DROP OPERATOR in SQL92.

DROP RULE

Name

DROP RULE Removes an existing rule from the database

```
DROP RULE name
```

Inputs

```
name
    The name of an existing rule to drop.
```

Outputs

```
DROP
    Message returned if successfully.
ERROR: RewriteGetRuleEventRel: rule "name" not found
    This message occurs if the specified rule does not exist.
```

Description

DROP RULE drops a rule from the specified Postgres rule system. Postgres will immediately cease enforcing it and will purge its definition from the system catalogs.

Notes

The DROP RULE statement is a Postgres language extension.

Refer to CREATE RULE for information on how to create rules.

Bugs

Once a rule is dropped, access to historical information the rule has written may disappear.

Usage

To drop the rewrite rule newrule:

```
DROP RULE newrule
```

Compatibility

SQL92

There is no DROP RULE in SQL92.

DROP SEQUENCE

Name

DROP SEQUENCE Removes an existing sequence

```
DROP SEQUENCE seqname [, ...]
```

Inputs

`seqname`

The name of a sequence.

Outputs

`DROP`

The message returned if the sequence is successfully dropped.

`WARN: Relation "seqname" does not exist.`

This message occurs if the specified sequence does not exist.

Description

DROP SEQUENCE removes sequence number generators from the data base. With the current implementation of sequences as special tables it works just like the DROP TABLE statement.

Notes

The DROP SEQUENCE statement is a Postgres language extension.

Refer to the CREATE SEQUENCE statement for information on how to create a sequence.

Usage

To remove sequence serial from database:

```
DROP SEQUENCE serial
```

Compatibility

SQL92

There is no DROP SEQUENCE in SQL92.

DROP TABLE

Name

DROP TABLE Removes existing tables from a database

```
DROP TABLE table [, ...]
```

Inputs

`table`
: The name of an existing table or view to drop.

Outputs

`DROP`
: The message returned if the command completes successfully.

`ERROR Relation "table" Does Not Exist!`
: If the specified table or view does not exist in the database.

Description

DROP TABLE removes tables and views from the database. Only its owner may destroy a table or view. A table may be emptied of rows, but not destroyed, by using DELETE.

If a table being destroyed has secondary indexes on it, they will be removed first. The removal of just a secondary index will not affect the contents of the underlying table.

Notes

Refer to CREATE TABLE and ALTER TABLE for information on how to create or modify tables.

Usage

To destroy the films and distributors tables:

```
DROP TABLE films, distributors
```

Compatibility

SQL92

SQL92 specifies some additional capabilities for DROP TABLE:

```
DROP TABLE table { RESTRICT | CASCADE }
```

RESTRICT

Ensures that only a table with no dependent views or integrity constraints can be destroyed.

CASCADE

Any referencing views or integrity constraints will also be dropped.

Tip: At present, to remove a referenced view you must drop it explicitly.

DROP TRIGGER

Name

DROP TRIGGER Removes the definition of a trigger

```
DROP TRIGGER name ON table
```

Inputs

```
name
    The name of an existing trigger.
table
    The name of a table.
```

Outputs

```
DROP
    The message returned if the trigger is successfully
   dropped.
ERROR: DropTrigger: there is no trigger name on relation
"table"
    This message occurs if the trigger specified does not
   exist.
```

Description

DROP TRIGGER will remove all references to an existing trigger definition. To execute this command the current user must be the owner of the trigger.

Notes

DROP TRIGGER is a Postgres language extension.

Refer to CREATE TRIGGER for information on how to create triggers.

Usage

Destroy the if_dist_exists trigger on table films:

```
DROP TRIGGER if_dist_exists ON films;
```

Compatibility

SQL92

There is no DROP TRIGGER statement in SQL92.

DROP TYPE

Name

DROP TYPE Removes a user-defined type from the system catalogs

```
DROP TYPE typename
```

Inputs

```
typename
    The name of an existing type.
```

Outputs

```
DROP
    The message returned if the command is successful.
ERROR: RemoveType: type 'typename' does not exist
    This message occurs if the specified type is not found.
```

Description

DROP TYPE will remove a user type from the system catalogs.

Only the owner of a type can remove it.

Notes

DROP TYPE statement is a Postgres language extension.

Refer to CREATE TYPE for inforamation on how to create types.

It is the user's responsibility to remove any operators, functions, aggregates, access methods, subtypes, and classes that use a deleted type.

Bugs

If a built-in type is removed, the behavior of the backend is unpredictable.

Usage

To remove the box type:

```
DROP TYPE box
```

Compatibility

SQL3

DROP TYPE is a SQL3 statement.

DROP USER

Name

DROP USER Removes an user account information

```
DROP USER username
```

Inputs

```
username
    The name of an existing user.
```

Outputs

```
DROP
    The message returned if the user is successfully deleted.
ERROR: removeUser: user "username" does not exist.
    This message occurs if the username is not found.
```

Description

DROP USER removes the specified user from the database, along with any databases owned by the user. It does not remove tables, views, or triggers owned by the named user in databases not owned by the user. This statement can be used in place of the destroyuser script, regardless of how the user was created.

Notes

DROP USER is a Postgres language extension.

Refer to CREATE USER and ALTER USER for information on how to create or modify user accounts.

Usage

To drop a user account:

```
DROP USER Jonathan;
```

Compatibility

SQL92

There is no DROP USER in SQL92.

DROP VIEW

Name

DROP VIEW Removes an existing view from a database

```
DROP VIEW view
```

Inputs

```
view
    The name of an existing view.
```

Outputs

```
DROP
    The message returned if the command is successful.
ERROR: RewriteGetRuleEventRel: rule "_RETview" not found
   This message occurs if the specified view does not exist in
   the database.
```

Description

DROP VIEW drops an existing view from the database. To execute this command you must be the owner of the view.

Notes

The Postgres DROP TABLE statement also drops views.

Refer to CREATE VIEW for information on how to create views.

Usage

This command will remove the view called kinds:

```
DROP VIEW kinds;
```

Compatibility

SQL92

SQL92 specifies some additional capabilities for DROP VIEW:

```
DROP VIEW view { RESTRICT | CASCADE }
```

Inputs

RESTRICT

Ensures that only a view with no dependent views or integrity constraints can be destroyed.

CASCADE

Any referencing views and integrity constraints will be dropped as well.

Notes

Tip: At present, to remove a referenced view from a Postgres database, you must drop it explicitly.

EXPLAIN

Name

EXPLAIN Shows statement execution details

```
EXPLAIN [ VERBOSE ] query
```

Inputs

VERBOSE
   Flag to show detailed query plan.

query
   Any query.

Outputs

NOTICE: QUERY PLAN: plan
   Explicit query plan from the Postgres backend.

EXPLAIN
   Flag sent after query plan is shown.

Description

This command outputs details about the supplied query. The default output is the computed query cost. VERBOSE displays the full query plan and cost to your screen, and pretty-prints the plan to the postmaster log file.

Notes

There is only sparse documentation on the optimizer's use of cost information in Postgres. General information on cost estimation for query optimization can be found in database textbooks. Refer to the Programmer's Guide in the chapters on indexes and the genetic query optimizer for more information.

Usage

To show a query plan for a simple query:

```
postgres=> explain select * from foo;
NOTICE:  QUERY PLAN:

Seq Scan on foo  (cost=0.00 size=0 width=4)

EXPLAIN
```

Compatibility

SQL92

There is no EXPLAIN statement defined in SQL92.

FETCH

Name

FETCH Gets rows using a cursor

```
FETCH [ selector ] [ count ]
    { IN | FROM } cursor
FETCH [ RELATIVE ] [ { [ # | ALL | NEXT | PRIOR ] } ] FROM ] cursor
```

Inputs

selector

selector defines the fetch direction. It can be one the following:

FORWARD

fetch next row(s). This is the default if selector is omitted.

BACKWARD

fetch previous row(s).

RELATIVE

Noise word for SQL92 compatibility.

count

count determines how many rows to fetch. It can be one of the following:

#

A signed integer that specify how many rows to fetch. Note that a negative integer is equivalent to changing the sense of FORWARD and BACKWARD.

ALL

Retrieve all remaining rows.

NEXT

Equivalent to specifying a count of 1.

PRIOR

Equivalent to specifying a count of -1.

cursor

An open cursor’s name.

Outputs

FETCH returns the results of the query defined by the specified cursor. The following messages will be returned if the query fails:

NOTICE: PerformPortalFetch: portal "cursor" not found

If cursor is not previously declared. The cursor must be declared within a transaction block.

NOTICE: FETCH/ABSOLUTE not supported, using RELATIVE

Postgres does not support absolute positioning of cursors.

ERROR: FETCH/RELATIVE at current position is not supported

```
SQL92 allows one to repetatively retrieve the cursor at its
"current position" using the syntax
FETCH RELATIVE 0 FROM cursor
Postgres does not currently support this notion; in fact the
value zero is reserved to indicate that all rows should be
retrieved and is equivalent to specifying the ALL keyword.
If the RELATIVE keyword has been used, the Postgres assumes
that the user intended SQL92 behavior and returns this error
message.
```

Description

FETCH allows a user to retrieve rows using a cursor. The number of rows retrieved is specified by #. If the number of rows remaining in the cursor is less than #, then only those available are fetched. Substituting the keyword ALL in place of a number will cause all remaining rows in the cursor to be retrieved. Instances may be fetched in both FORWARD and BACKWARD directions. The default direction is FORWARD.

> Tip: Negative numbers are now allowed to be specified for the row count. A negative number is equivalent to reversing the sense of the FORWARD and BACKWARD keywords. For example, FORWARD -1 is the same as BACKWARD 1.

Note that the FORWARD and BACKWARD keywords are Postgres extensions. The SQL92 syntax is also supported, specified in the second form of the command. See below for details on compatibility issues.

Once all rows are fetched, every other fetch access returns no rows.

Updating data in a cursor is not supported by Postgres, because mapping cursor updates back to base tables is not generally possible, as is also the case with VIEW updates. Consequently, users must issue explicit UPDATE commands to replace data.

Cursors may only be used inside of transactions because the data that they store spans multiple user queries.

Notes

Refer to MOVE statements to change cursor position. Refer to DECLARE statements to declare a cursor. Refer to BEGIN WORK, COMMIT WORK, ROLLBACK WORK statements for further information about transactions.

Usage

```
--set up and use a cursor:
--
BEGIN WORK;
  DECLARE liahona CURSOR
     FOR SELECT * FROM films;

--Fetch first 5 rows in the cursor liahona:
--
  FETCH FORWARD 5 IN liahona;

  code |title                 |did| date_prod|kind      |len
  -----+----------------------+---+----------+----------+------
  BL101|The Third Man         |101|1949-12-23|Drama     | 01:44
  BL102|The African Queen     |101|1951-08-11|Romantic  | 01:43
  JL201|Une Femme est une Femme|102|1961-03-12|Romantic  | 01:25
  P_301|Vertigo               |103|1958-11-14|Action    | 02:08
  P_302|Becket                |103|1964-02-03|Drama     | 02:28
```

```
--Fetch previous row:
--
  FETCH BACKWARD 1 IN liahona;

  code |title                 |did| date_prod|kind      |len
  -----+----------------------+---+----------+----------+------
  P_301|Vertigo               |103|1958-11-14|Action    | 02:08

-- close the cursor and commit work:
--
  CLOSE liahona;
COMMIT WORK;
```

Compatibility

The non-embedded use of cursors is a Postgres extension. The syntax and usage of cursors is being compared against the embedded form of cursors defined in SQL92.

SQL92

SQL92 allows absolute positioning of the cursor for FETCH, and allows placing the results into explicit variables.

```
FETCH ABSOLUTE #
    FROM cursor
    INTO :variable [, ...]
```

ABSOLUTE

The cursor should be positioned to the specified absolute row number. All row numbers in Postgres are relative numbers so this capability is not supported.

:variable

Target host variable(s).

GRANT

Name

GRANT Grants access privilege to a user, a group or all users

```
GRANT privilege [, ...]
    ON object [, ...]
    TO { PUBLIC | GROUP group | username }
```

Inputs

privilege

The possible privileges are:

SELECT

Access all of the columns of a specific table/view.

INSERT

Insert data into all columns of a specific table.

UPDATE

Update all columns of a specific table.

DELETE

Delete rows from a specific table.

RULE

Define rules on the table/view (See CREATE RULE statement).

ALL

Grant all privileges.

object

The name of an object to which to grant access. The possible objects are:

table
view
sequence
index

PUBLIC

A short form representing all users.

GROUP group

A group to whom to grant privileges. In the current release, the group must be created explicitly as described below.

username

The name of a user to whom grant privileges. PUBLIC is a short form representing all users.

Outputs

CHANGE

Message returned if successful.

ERROR: ChangeAcl: class "object" not found

Message returned if the specified object is not available or if it is impossible to give privileges to the specified group or users.

Description

GRANT allows the creator of an object to give specific permissions to all users (PUBLIC) or to a certain user or group. Users other than the creator don't have any access permission unless the creator GRANTs permissions, after the object is created.

Once a user has a privilege on an object, he is enabled to exercise that privilege. There is no need to GRANT privileges to the creator of an object, the creator automatically holds ALL privileges, and can also drop the object.

Notes

Use the psql \z command for further information about permissions on existing objects:

```
Database    = lusitania
+------------------+-----------------------------------------------+
|  Relation        |        Grant/Revoke Permissions               |
+------------------+-----------------------------------------------+
| mytable          | {"=rw","miriam=arwR","group todos=rw"}        |
+------------------+-----------------------------------------------+
Legend:
      uname=arwR -- privileges granted to a user
group gname=arwR -- privileges granted to a GROUP
           =arwR -- privileges granted to PUBLIC

               r -- SELECT
               w -- UPDATE/DELETE
               a -- INSERT
               R -- RULE
            arwR -- ALL
```

Tip: Currently, to create a GROUP you have to insert data manually into table pg_group as:

```
INSERT INTO pg_group VALUES ('todos');
CREATE USER miriam IN GROUP todos;
```

Refer to REVOKE statements to revoke access privileges.

Usage

```
-- grant insert privilege to all users on table films:
--
GRANT INSERT ON films TO PUBLIC;

-- grant all privileges to user manuel on view kinds:
--
GRANT ALL ON kinds TO manuel;
```

Compatibility

SQL92

The SQL92 syntax for GRANT allows setting privileges for individual columns within a table, and allows setting a privilege to grant the same privileges to others.

```
GRANT privilege [, ...]
    ON object [ ( column [, ...] ) ] [, ...]
    TO { PUBLIC | username [, ...] }
    [ WITH GRANT OPTION ]
```

Fields are compatible with the those in the Postgres implementation, with the following additions:

privilege SELECT

SQL92 permits additional privileges to be specified:

REFERENCES

Allowed to reference some or all of the columns of a specific table/view in integrity constraints.

USAGE

Allowed to use a domain, character set, collation or translation. If an object specifies anything other than a table/view, privilege must specify only USAGE.

Tip: Currently, to grant privileges in Postgres to only few columns, you must create a view having desired columns and then grant privileges to that view.

object

object

SQL92 allows an additional non-functional keyword:

[TABLE] table

CHARACTER SET

Allowed to use the specified character set.

COLLATION

Allowed to use the specified collation sequence.

TRANSLATION

Allowed to use the specified character set translation.

DOMAIN

Allowed to use the specified domain.

WITH GRANT OPTION

Allowed to grant the same privilege to others.

INSERT

Name

INSERT Inserts new rows into a table

```
INSERT INTO table [ ( column [, ...] ) ]
    { VALUES ( expression [, ...] ) | SELECT query }
```

Inputs

table

The name of an existing table.

column

The name of a column in table.

expression

A valid expression or value to assign to column.

query

A valid query. Refer to the SELECT statement for a further description of valid arguments.

Outputs

INSERT oid 1

Message returned if only one row was inserted. oid is the row identifier.

INSERT 0 #

Message returned if more than one rows were inserted. # is the number of rows inserted.

Description

INSERT allows one to insert new rows into a table. One can insert a single row at time or several rows as a result of a query. The columns in the target list may be listed in any order. In every column not present in the target list will be inserted the default value, if column has not a declared default value it will be assumed as NULL. If the expression for each column is not of the correct data type, automatic type coercion will be attempted.

You must have insert privilege to a table in order to append to it, as well as select privilege on any table specified in a WHERE clause.

Usage

```
--Insert a single row into table films;
--(in the second example the column date_prod is omitted
--therefore will be stored in it a default value of NULL):
--
INSERT INTO films VALUES
    ('UA502','Bananas',105,'1971-07-13','Comedy',INTERVAL '82 minute');

INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DATE '1961-06-16', 'Drama');

--Insert a single row into table distributors, note that
--only column "name" is specified, to the non specified
--column "did" will be assigned its default value:
```

```
--
INSERT INTO distributors (name) VALUES ('British Lion');

--Insert several rows into table films from table tmp:
--
INSERT INTO films
    SELECT * FROM tmp;

--Insert into arrays:
--Create an empty 3x3 gameboard for noughts-and-crosses
--(all of these queries create the same board attribute)
--(Refer to the PostgreSQL User's Guide for further
--information about arrays).

INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1,'{{"","",""},{},{"",""}}');
INSERT INTO tictactoe (game, board[3][3])
    VALUES (2,'{}');
INSERT INTO tictactoe (game, board)
    VALUES (3,'{{,,},{,,},{,,}}');
```

Compatibility

SQL92

The INSERT statement is fully compatible with SQL92. Possible limitations in features of the query clause are documented for the SELECT statement.

LISTEN

Name

LISTEN Listen for notification on a notify condition

LISTEN notifyname

Inputs

notifyname

Name of notify condition.

Outputs

LISTEN

Message returned upon successful completion of registration.

NOTICE Async_Listen: We are already listening on notifyname

If this backend is already registered for that notify condition.

Description

LISTEN registers the current Postgres backend as a listener on the notify condition notifyname.

Whenever the command NOTIFY notifyname is invoked, either by this backend or another one connected to the same database, all the backends currently listening on that notify condition are notified, and each will in turn notify its connected frontend application. See the discussion of NOTIFY for more information.

A backend can be deregistered for a given notify condition with the UNLISTEN command. Also, a backend's listen registrations are automatically cleared when the backend process exits.

The method a frontend application must use to detect notify events depends on which Postgres application programming interface it uses. With the basic libpq library, the application issues LISTEN as an ordinary SQL command, and then must periodically call the routine PQnotifies to find out whether any notify events have been received. Other interfaces such as libpgtcl provide higher-level methods for handling notify events; indeed, with libpgtcl the application programmer should not even issue LISTEN or UNLISTEN directly. See the documentation for the library you are using for more details.

The reference page for NOTIFY contains a more extensive discussion of the use of LISTEN and NOTIFY.

Notes

notifyname can be any string valid as a name; it need not correspond to the name of any actual table. If notifyname is enclosed in double-quotes, it need not even be a syntactically valid name, but can be any string up to 31 characters long.

In some previous releases of Postgres, notifyname had to be enclosed in double-quotes when it did not correspond to any existing table name, even if syntactically valid as a name. That is no longer required.

Usage

```
-- Configure and execute a listen/notify sequence from psql
```

```
postgres=> listen virtual;
LISTEN
postgres=> notify virtual;
NOTIFY
ASYNC NOTIFY of 'virtual' from backend pid '11239' received
```

Compatibility

SQL92

There is no LISTEN in SQL92.

LOAD

Name

LOAD Dynamically loads an object file

```
LOAD 'filename'
```

Inputs

filename

Object file for dynamic loading.

Outputs

LOAD

Message returned on successful completion.

ERROR: LOAD: could not open file 'filename'

Message returned if the specified file is not found. The file must be visible to the Postgres backend, with the appropriate full path name specified, to avoid this message.

Description

Loads an object (or ".o") file into the Postgres backend address space. Once a file is loaded, all functions in that file can be accessed. This function is used in support of user-defined types and functions.

If a file is not loaded using LOAD, the file will be loaded automatically the first time the function is called by Postgres. LOAD can also be used to reload an object file if it has been edited and recompiled. Only objects created from C language files are supported at this time.

Notes

Functions in loaded object files should not call functions in other object files loaded through the LOAD command. For example, all functions in file A should call each other, functions in the standard or math libraries, or in Postgres itself. They should not call functions defined in a different loaded file B. This is because if B is reloaded, the Postgres loader is not able to relocate the calls from the functions in A into the new address space of B. If B is not reloaded, however, there will not be a problem.

Object files must be compiled to contain position independent code. For example, on DECstations you must use /bin/cc with the -G 0 option when compiling object files to be loaded.

Note that if you are porting Postgres to a new platform, LOAD will have to work in order to support ADTs.

Usage

```
--Load the file /usr/postgres/demo/circle.o
--
```

```
LOAD '/usr/postgres/demo/circle.o'
```

Compatibility

SQL92

There is no LOAD in SQL92.

LOCK

Name

LOCK Explicit lock of a table inside a transaction

LOCK [TABLE] table

Inputs

table

The name of an existing table to lock.

Outputs

DELETE 0

Message returned on a successful lock. LOCK is implemented as a DELETE FROM table which is guaranteed to not delete any rows.

ERROR table: Table does not exist.

Message returned if table does not exist.

Description

LOCK locks in exclusive mode a table inside a transaction. The classic use for this is the case where you want to select some data, then update it inside a transaction. If you don't explicit lock a table using LOCK statement, it will be implicit locked only at the first UPDATE, INSERT, or DELETE operation. If you don't exclusive lock the table before the select, some other user may also read the selected data, and try and do their own update, causing a deadlock while you both wait for the other to release the select-induced shared lock so you can get an exclusive lock to do the update.

Another example of deadlock is where one user locks one table, and another user locks a second table. While both keep their existing locks, the first user tries to lock the second user's table, and the second user tries to lock the first user's table. Both users deadlock waiting for the tables to become available. The only solution to this is for both users to lock tables in the same order, so user's lock acquisitions and requests to not form a deadlock.

Note: Postgres does detect deadlocks and will rollback transactions to resolve the deadlock. Usually, at least one of the deadlocked transactions will complete successfully.

Notes

LOCK is a Postgres language extension.

LOCK works only inside transactions.

Bug: If the locked table is dropped then it will be automatically unlocked even if a transaction is still in progress.

Usage

```
--Explicit locking to prevent deadlock:
--
BEGIN WORK;
    LOCK films;
    SELECT * FROM films;
```

```
    UPDATE films SET len = INTERVAL '100 minute'
        WHERE len = INTERVAL '117 minute';
COMMIT WORK;
```

Compatibility

SQL92

There is no LOCK TABLE in SQL92, which instead uses SET TRANSACTION to specify concurrency level on transactions.

MOVE

Name

MOVE Moves cursor position

```
MOVE [ selector ] [ count ]
    { IN | FROM } cursor
FETCH [ RELATIVE ] [ { [ # | ALL | NEXT | PRIOR ] } ] FROM ] cursor
```

Description

MOVE allows a user to move cursor position a specified number of rows. MOVE works like the FETCH command, but only positions the cursor and does not return rows.

Refer to the FETCH command for details on syntax and usage.

Notes

MOVE is a Postgres language extension.

Refer to FETCH for a description of valid arguments. Refer to DECLARE to declare a cursor. Refer to BEGIN WORK, COMMIT WORK, ROLLBACK WORK statements for further information about transactions.

Usage

```
--set up and use a cursor:
--
BEGIN WORK;
    DECLARE liahona CURSOR  FOR SELECT * FROM films;

    --Skip first 5 rows:
    --
    MOVE FORWARD 5 IN liahona;
MOVE
    --Fetch 6th row in the cursor liahona:
    --
    FETCH 1 IN liahona;
FETCH
code |title |did| date_prod|kind      |len
-----+------+---+----------+----------+------
P_303|48 Hrs|103|1982-10-22|Action    | 01:37
(1 row)
    -- close the cursor liahona and commit work:
    --
    CLOSE liahona;
COMMIT WORK;
```

Compatibility

SQL92

There is no SQL92 MOVE statement. Instead, SQL92 allows one to FETCH rows from an absolute cursor position, implicitly moving the cursor to the correct place.

NOTIFY

Name

NOTIFY Signals all frontends and backends listening on a notify condition

```
NOTIFY notifyname
```

Inputs

`notifyname`
    Notify condition to be signaled.

Outputs

`NOTIFY`
    Acknowledgement that notify command has executed.

`Notify events`
    Events are delivered to listening frontends; whether and how each frontend application reacts depends on its programming.

Description

The NOTIFY command sends a notify event to each frontend application that has previously executed LISTEN notifyname for the specified notify condition in the current database.

The information passed to the frontend for a notify event includes the notify condition name and the notifying backend process's PID. It is up to the database designer to define the condition names that will be used in a given database and what each one means.

Commonly, the notify condition name is the same as the name of some table in the database, and the notify event essentially means "I changed this table, take a look at it to see what's new". But no such association is enforced by the NOTIFY and LISTEN commands. For example, a database designer could use several different condition names to signal different sorts of changes to a single table.

NOTIFY provides a simple form of signal or IPC (interprocess communication) mechanism for a collection of processes accessing the same Postgres database. Higher-level mechanisms can be built by using tables in the database to pass additional data (beyond a mere condition name) from notifier to listener(s).

When NOTIFY is used to signal the occurrence of changes to a particular table, a useful programming technique is to put the NOTIFY in a rule that is triggered by table updates. In this way, notification happens automatically when the table is changed, and the application programmer can't accidentally forget to do it.

NOTIFY interacts with SQL transactions in some important ways. Firstly, if a NOTIFY is executed inside a transaction, the notify events are not delivered until and unless the transaction is committed. This is appropriate, since if the transaction is aborted we would like all the commands within it to have had no effect --- including NOTIFY. But it can be disconcerting if one is expecting the notify events to be delivered immediately. Secondly, if a listening backend receives a notify signal while it is within a transaction, the notify event will not be delivered to its connected frontend until just after the transaction is completed (either committed or aborted). Again, the reasoning is that if a notify were delivered within a transaction that was later aborted, one would want the notification to be undone somehow --- but the backend cannot "take back" a notify once it has sent it to the frontend. So notify events are only

delivered between transactions. The upshot of this is that applications using NOTIFY for real-time signaling should try to keep their transactions short.

NOTIFY behaves like Unix signals in one important respect: if the same condition name is signaled multiple times in quick succession, recipients may get only one notify event for several executions of NOTIFY. So it is a bad idea to depend on the number of notifies received. Instead, use NOTIFY to wake up applications that need to pay attention to something, and use a database object (such as a sequence) to keep track of what happened or how many times it happened.

It is common for a frontend that sends NOTIFY to be listening on the same notify name itself. In that case it will get back a notify event, just like all the other listening frontends. Depending on the application logic, this could result in useless work --- for example, re-reading a database table to find the same updates that that frontend just wrote out. In Postgres 6.4 and later, it is possible to avoid such extra work by noticing whether the notifying backend process's PID (supplied in the notify event message) is the same as one's own backend's PID (available from libpq). When they are the same, the notify event is one's own work bouncing back, and can be ignored. (Despite what was said in the preceding paragraph, this is a safe technique. Postgres keeps self-notifies separate from notifies arriving from other backends, so you cannot miss an outside notify by ignoring your own notifies.)

Notes

notifyname can be any string valid as a name; it need not correspond to the name of any actual table. If notifyname is enclosed in double-quotes, it need not even be a syntactically valid name, but can be any string up to 31 characters long.

In some previous releases of Postgres, notifyname had to be enclosed in double-quotes when it did not correspond to any existing table name, even if syntactically valid as a name. That is no longer required.

In Postgres releases prior to 6.4, the backend PID delivered in a notify message was always the PID of the frontend's own backend. So it was not possible to distinguish one's own notifies from other clients' notifies in those earlier releases.

Usage

```
-- Configure and execute a listen/notify sequence from psql
postgres=> listen virtual;
LISTEN
postgres=> notify virtual;
NOTIFY
ASYNC NOTIFY of 'virtual' from backend pid '11239' received
```

Compatibility

SQL92

There is no NOTIFY statement in SQL92.

RESET

Name

RESET Restores run-time parameters for session to default values

RESET variable

Inputs

variable

Refer to the SET statement for more information on available variables.

Outputs

RESET VARIABLE

Message returned if variable is successfully reset to its default value..

Description

RESET restores variables to the default values. Refer to the SET command for details on allowed values and defaults. RESET is an alternate form for

SET variable = DEFAULT

Notes

The RESET statement is a Postgres language extension.

Refer to SET/SHOW statements to set/show variable values.

Usage

```
-- reset DateStyle to its default;
RESET DateStyle;


-- reset Geqo to its default;
RESET GEQO;
```

Compatibility

SQL92

There is no RESET in SQL92.

REVOKE

Name

REVOKE Revokes access privilege from a user, a group or all users.

```
REVOKE privilege [, ...]
    ON object [, ...]
    FROM { PUBLIC | GROUP group | username }
```

Inputs

privilege

The possible privileges are:

SELECT

Privilege to access all of the columns of a specific table/view.

INSERT

Privilege to insert data into all columns of a specific table.

UPDATE

Privilege to update all columns of a specific table.

DELETE

Privilege to delete rows from a specific table.

RULE

Privilege to define rules on table/view. (See CREATE RULE).

ALL

Rescind all privileges.

object

The name of an object from which to revoke access. The possible objects are:

table
view
sequence
index

group

The name of a group from whom to revoke privileges.

username

The name of a user from whom revoke privileges. Use the PUBLIC keyword to specify all users.

PUBLIC

Rescind the specified privilege(s) for all users.

Outputs

CHANGE

Message returned if successfully.

ERROR

Message returned if object is not available or impossible to revoke privileges from a group or users.

Description

REVOKE allows creator of an object to revoke permissions granted before, from all users (via PUBLIC) or a certain user or group.

Notes

Refer to psql \z command for further information about permissions on existing objects:

```
Database    = lusitania
+------------------+---------------------------------------------+
|  Relation        |        Grant/Revoke Permissions             |
+------------------+---------------------------------------------+
| mytable          | {"=rw","miriam=arwR","group todos=rw"}      |
+------------------+---------------------------------------------+
Legend:
      uname=arwR -- privileges granted to a user
group gname=arwR -- privileges granted to a GROUP
           =arwR -- privileges granted to PUBLIC

               r -- SELECT
               w -- UPDATE/DELETE
               a -- INSERT
               R -- RULE
            arwR -- ALL
```

Tip: Currently, to create a GROUP you have to insert data manually into table pg_group as:

```
INSERT INTO pg_group VALUES ('todos');
CREATE USER miriam IN GROUP todos;
```

Usage

```
-- revoke insert privilege from all users on table films:
--
REVOKE INSERT ON films FROM PUBLIC;

-- revoke all privileges from user manuel on view kinds:
--
```

```
REVOKE ALL ON kinds FROM manuel;
```

Compatibility

SQL92

The SQL92 syntax for REVOKE has additional capabilities for rescinding privileges, including those on individual columns in tables:

```
REVOKE { SELECT | DELETE | USAGE | ALL PRIVILEGES } [, ...]
    ON object
    FROM { PUBLIC | username [, ...] } { RESTRICT | CASCADE }
REVOKE { INSERT | UPDATE | REFERENCES } [, ...] [ ( column [, ...] ) ]
    ON object
    FROM { PUBLIC | username [, ...] } { RESTRICT | CASCADE }
```

Refer to the GRANT command for details on individual fields.

```
REVOKE GRANT OPTION FOR privilege [, ...]
    ON object
    FROM { PUBLIC | username [, ...] } { RESTRICT | CASCADE }
```

Rescinds authority for a user to grant the specified privilege to others. Refer to the GRANT command for details on individual fields.

The possible objects are:

[TABLE] table/view
CHARACTER SET character-set
COLLATION collation
TRANSLATION translation
DOMAIN domain

If user1 gives a privilege WITH GRANT OPTION to user2, and user2 gives it to user3 then user1 can revoke this privilege in cascade using the CASCADE keyword.

If user1 gives a privilege WITH GRANT OPTION to user2, and user2 gives it to user3 then if user1 try revoke this privilege it fails if he/she specify the RESTRICT keyword.

ROLLBACK

Name

ROLLBACK Aborts the current transaction

```
ROLLBACK [ WORK ]
```

Inputs

```
None.
```

Outputs

```
 ABORT
    Message returned if successful.
NOTICE: UserAbortTransactionBlock and not in in-progress state
ABORT
    If there is not any transaction currently in progress.
```

Description

ROLLBACK rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Notes

The keyword WORK is noise and can be omitted.

Use the COMMIT statement to successfully terminate a transaction.

Usage

```
--To abort all changes:
--
ROLLBACK WORK;
```

Compatibility

SQL92

Full compatibility.

SELECT

Name

SELECT Retrieve rows from a table or view.

```
SELECT [ALL|DISTINCT [ON column] ]
    expression [ AS name ] [, ...]
    [ INTO [TABLE] new_table ]
    [ FROM table [alias ] [, ...] ]
    [ WHERE condition ]
    [ GROUP BY column [, ...] ]
    [ HAVING condition [, ...] ]
    [ UNION [ALL] select ]
    [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

Inputs

expression

The name of a table’s column or an expression.

name

Specifies another name for a column or an expression using the AS clause. name cannot be used in the WHERE condition. It can, however, be referenced in associated ORDER BY or GROUP BY clauses.

new_table

If the INTO TABLE clause is specified, the result of the query will be stored in another table with the indicated

name. If new_table does not exist, it will be created automatically. Refer to SELECT INTO for more information.

Note: The CREATE TABLE AS statement will also create a new table from a select query.

table
: The name of an existing table referenced by the FROM clause.

alias
: An alternate name for the preceding table. It is used for brevity or to eliminate ambiguity for joins within a single table.

condition
: A boolean expression giving a result of true or false. See the WHERE clause.

column
: The name of a table's column.

select
: A select statement with all features except the ORDER BY clause.

Outputs

Rows
: The complete set of rows resulting from the query specification.

count
: The count of rows returned by the query.

Description

SELECT will get all rows which satisfy the WHERE condition or all rows of a table if WHERE is omitted.

The GROUP BY clause allows a user to divide a table conceptually into groups. (See GROUP BY clause).

The HAVING clause specifies a grouped table derived by the elimination of groups from the result of the previously specified clause. (See HAVING clause).

The ORDER BY clause allows a user to specify that he/she wishes the rows sorted according to the ASCending or DESCending mode operator. (See ORDER BY clause)

The UNION clause specifies a table derived from a Cartesian product union join. (See UNION clause).

You must have SELECT privilege to a table to read its values (See GRANT/REVOKE statements).

WHERE Clause

The optional WHERE condition has the general form:

```
WHERE expr cond_op expr [ log_op ... ]
```

where cond_op can be one of: =, <, <=, >, >=, <> or a conditional operator like ALL, ANY, IN, LIKE, et cetera and log_op can be one of: AND, OR, NOT. The comparison returns either TRUE or FALSE and all instances will be discarded if the expression evaluates to FALSE.

GROUP BY Clause

GROUP BY specifies a grouped table derived by the application of the this clause:

```
GROUP BY column [, ...]
```

HAVING Clause

The optional HAVING condition has the general form:

```
HAVING cond_expr
```

where cond_expr is the same as specified for the WHERE clause.

HAVING specifies a grouped table derived by the elimination of groups from the result of the previously specified clause that do not meet the cond_expr.

Each column referenced in cond_expr shall unambiguously reference a grouping column.

ORDER BY Clause

```
ORDER BY column [ ASC | DESC ] [, ...]
```

column can be either a column name or an ordinal number.

The ordinal numbers refers to the ordinal (left-to-right) position of the column. This feature makes it possible to define an ordering on the basis of a column that does not have a proper name. This is never absolutely necessary because it is always possible assign a name to a calculated column using the AS clause, e.g.:

```
SELECT title, date_prod + 1 AS newlen FROM films ORDER BY newlen;
```

The columns in the ORDER BY must appear in the SELECT clause. Thus the following statement is illegal:

```
SELECT name FROM distributors ORDER BY code;
```

Optionally one may add the keyword DESC (descending) or ASC (ascending) after each column name in the ORDER BY clause. If not specified, ASC is assumed by default.

UNION Clause

```
table_query UNION [ ALL ] table_query
     [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

where table_query specifies any select expression without an ORDER BY clause.

The UNION operator specifies a table derived from a Cartesian product. The two tables that represent the direct operands of the UNION must have the same number of columns, and corresponding columns must be of compatible data types.

By default, the result of UNION does not contain any duplicate rows unless the ALL clause is specified.

Multiple UNION operators in the same SELECT statement are evaluated left to right. Note that the ALL keyword is not global in nature, being applied only for the current pair of table results.

Usage

To join the table films with the table distributors:

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d, films f
    WHERE f.did = d.did
```

```
    title                    |did|name             | date_prod|kind
    -------------------------+---+-----------------+----------+---------
--
    The Third Man            |101|British Lion     |1949-12-23|Drama
    The African Queen        |101|British Lion     |1951-08-11|Romantic
    Une Femme est une Femme  |102|Jean Luc Godard  |1961-03-12|Romantic
    Vertigo                  |103|Paramount        |1958-11-14|Action
    Becket                   |103|Paramount        |1964-02-03|Drama
    48 Hrs                   |103|Paramount        |1982-10-22|Action
    War and Peace            |104|Mosfilm          |1967-02-12|Drama
    West Side Story          |105|United Artists   |1961-01-03|Musical
    Bananas                  |105|United Artists   |1971-07-13|Comedy
    Yojimbo                  |106|Toho             |1961-06-16|Drama
    There's a Girl in my Soup|107|Columbia         |1970-06-11|Comedy
    Taxi Driver              |107|Columbia         |1975-05-15|Action
    Absence of Malice        |107|Columbia         |1981-11-15|Action
    Storia di una donna      |108|Westward         |1970-08-15|Romantic
    The King and I           |109|20th Century Fox|1956-08-11|Musical
    Das Boot                 |110|Bavaria Atelier  |1981-11-11|Drama
    Bed Knobs and Broomsticks|111|Walt Disney      |          |Musical
```

To sum the column len of all films and group the reults by kind:

```
SELECT kind, SUM(len) AS total FROM films GROUP BY kind;
```

```
    kind      |total
    ----------+------
    Action    | 07:34
    Comedy    | 02:58
    Drama     | 14:28
    Musical   | 06:42
    Romantic  | 04:38
```

To sum the column len of all films, group the reults by kind and show those group totals that are less than 5 hours:

```
SELECT kind, SUM(len) AS total
    FROM films
    GROUP BY kind
    HAVING SUM(len) < INTERVAL '5 hour';

    kind      |total
    ----------+------
    Comedy    | 02:58
    Romantic  | 04:38
```

The following two examples are identical ways of sorting the individual results according to the contents of the second column (name):

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;

    did|name
    ---+----------------
    109|20th Century Fox
    110|Bavaria Atelier
    101|British Lion
    107|Columbia
    102|Jean Luc Godard
    113|Luso films
    104|Mosfilm
    103|Paramount
    106|Toho
    105|United Artists
    111|Walt Disney
    112|Warner Bros.
    108|Westward
```

This example shows how to obtain the union of the tables distributors and actors, restricting the results to those that begin with letter W in each table. Only distinct rows are to be used, so the ALL keyword is omitted:

```
    --        distributors:               actors:
    --        did|name                    id|name
    --        ---+------------            --+--------------
    --        108|Westward                 1|Woody Allen
    --        111|Walt Disney              2|Warren Beatty
    --        112|Warner Bros.             3|Walter Matthau
    --        ...                         ...

SELECT distributors.name
    FROM   distributors
    WHERE  distributors.name LIKE 'W%'
UNION
SELECT actors.name
    FROM   actors
    WHERE  actors.name LIKE 'W%'

    name
    --------------
    Walt Disney
    Walter Matthau
    Warner Bros.
    Warren Beatty
    Westward
    Woody Allen
```

Compatibility

SQL92

SELECT Clause

In the SQL92 standard, the optional keyword "AS" is just noise and can be omitted without affecting the meaning. The Postgres parser requires this keyword when renaming columns because the type extensibility features lead to parsing ambiguities in this context.

In the SQL92 standard, the new column name specified in an "AS" clause may be referenced in GROUP BY and HAVING clauses. This is not currently allowed in Postgres.

UNION Clause

The SQL92 syntax for UNION allows an additional CORRESPONDING BY clause:

```
table_query UNION [ALL]
    [CORRESPONDING [BY (column [,...])]]
    table_query
```

The CORRESPONDING BY clause is not supported by Postgres.

SELECT INTO

Name

SELECT INTO Create a new table from an existing table or view

```
SELECT [ ALL | DISTINCT ] expression [ AS name ] [, ...]
    INTO [ TABLE ] new_table ]
    [ FROM table [alias] [, ...] ]
    [ WHERE condition ]
    [ GROUP BY column [, ...] ]
    [ HAVING condition [, ...] ]
    [ UNION [ ALL ] select]
    [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

Inputs

All input fields are described in detail for SELECT.

Outputs

All output fields are described in detail for SELECT.

Description

SELECT INTO creates a new table from the results of a query. Typically, this query draws data from an existing table, but any SQL query is allowed.

Note: CREATE TABLE AS is functionally equivalent to the SELECT INTO command.

SET

Name

SET Set run-time parameters for session

```
SET variable { TO | = } { 'value' | DEFAULT }
SET TIME ZONE { 'timezone' | LOCAL };
```

Inputs

variable

Settable global parameter.

value

New value of parameter.

The possible variables and allowed values are:

DateStyle

ISO
: use ISO 8601-style dates and times

SQL
: use Oracle/Ingres-style dates and times

Postgres
: use traditional Postgres format

European
: use dd/mm/yyyy for numeric date representations.

NonEuropean
: use mm/dd/yyyy for numeric date representations.

German
: use dd.mm.yyyy for numeric date representations.

US
: same as 'NonEuropean'

default
: restores the default values ('US,Postgres')

Date format initialization my be done by:

Setting PGDATESTYLE environment variable.
Running postmaster using -oe parameter to set dates to the 'European' convention. Note that this affects only the some combinations of date styles; for example the ISO style is not affected by this parameter.
Changing variables in src/backend/utils/init/globals.c.

The variables in globals.c which can be changed are:

```
bool EuroDates = false
                 true
int  DateStyle = USE_ISO_DATES
                 USE_POSTGRES_DATES
                 USE_ISO_DATES
```

USE_SQL_DATES
USE_GERMAN_DATES

TIMEZONE

The possible values for timezone depends on your operating system. For example on Linux /usr/lib/zoneinfo contains the database of timezones.

Here are some valid values for timezone:

'PST8PDT'

set the timezone for California

'Portugal'

set time zone for Portugal.

'Europe/Rome'

set time zone for Italy.

DEFAULT

set time zone to your local timezone (value of the TZ environment variable).

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

A frontend which uses libpq may be initialized by setting the PGTZ environment variable.

The second syntax shown above, allows one to set the timezone with a syntax similar to SQL92 SET TIME ZONE. The

LOCAL keyword is just an alternate form of DEFAULT for SQL92 compatibility.

There are also several internal or optimization parameters which can be specified by the SET command:

COST_HEAP

Sets the default cost of a heap scan for use by the optimizer.

float4

Set the cost of a heap scan to the specified floating point value.

DEFAULT

Sets the cost of a heap scan to the default value.

The frontend may be initialized by setting the PGCOSTHEAP environment variable.

COST_INDEX

Sets the default cost of an index scan for use by the optimizer.

float4

Set the cost of an index scan to the specified floating point value.

DEFAULT

Sets the cost of an index scan to the default value.

The frontend may be initialized by setting the PGCOSTINDEX environment variable.

GEQO

Sets the threshold for using the genetic optimizer algorithm.

On

enables the genetic optimizer algorithm for statements with 8 or more tables.

On=#

Takes an integer argument to enable the genetic optimizer algorithm for statements with # or more tables in the query.

Off

disables the genetic optimizer algorithm.

DEFAULT

Equivalent to specifying SET GEQO='on'

This algorithm is on by default, which used GEQO for statements of eight or more tables. (See the chapter on GEQO in the Programmer's Guide for more information).

The frontend may be initialized by setting PGGEQO environment variable.

R_PLANS

Determines whether right-hand plan evaluation is allowed:

On

enables right-hand evaluation of plans.

Off

disables right-hand evaluation of plans.

DEFAULT

Equivalent to specifying SET R_PLANS='off'.

It may be useful when joining big relations with small ones. This algorithm is off by default. It's not used by GEQO anyway.

The frontend may be initialized by setting the PGRPLANS environment variable.

KSQO

Key Set Query Optimizer forces the query optimizer to optimize repetative OR clauses such as generated by MicroSoft Access:

On

enables this optimization.

Off

disables this optimization.

DEFAULT

Equivalent to specifying SET KSQO='off'.

It may be useful when joining big relations with small ones. This algorithm is off by default. It's not used by GEQO anyway.

The frontend may be initialized by setting the PGRPLANS environment variable.

QUERY_LIMIT

Sets the number of rows returned by a query.

Value

```
       Maximum number of rows to return for a query. The default
       is to allow an unlimited number of rows.
#
       Sets the maximum number of rows returned by a query to #.
DEFAULT
       Sets the maximum number of rows returned by a query to be
       unlimited.
   By default, there is no limit to the number of rows returned
       by a query.
```

Outputs

```
SET VARIABLE
    Message returned if successfully.
WARN: Bad value for variable (value)
    If the command fails to set variable.
```

Description

SET will modify configuration parameters for variable during a session.

Current values can be obtained using SHOW, and values can be restored to the defaults using RESET. Parameters and values are case-insensitive. Note that the value field is always specified as a string, so is enclosed in single-quotes.

SET TIME ZONE changes the session's default time zone offset. A SQL-session always begins with an initial default time zone offset. The SET TIME ZONE statement is used to change the default time zone offset for the current SQL session.

Notes

The SET variable statement is a Postgres language extension.

Refer to SHOW and RESET to display or reset the current values.

Usage

```
--Set the style of date to ISO:
--
SET DATESTYLE TO 'ISO';


--Set GEQO to default:
--
SET GEQO = DEFAULT;


--Turn on right-hand evaluation of plans:
--
SET R_PLANS TO 'on';


--set the timezone for Berkeley, California:
SET TIME ZONE 'PST8PDT';

SELECT CURRENT_TIMESTAMP AS today;

   today
   ----------------------
```

```
   1998-03-31 07:41:21-08


--set the timezone for Italy:
SET TIME ZONE 'Europe/Rome';

SELECT CURRENT_TIMESTAMP AS today;

   today
   ----------------------
   1998-03-31 17:41:31+02
```

Compatibility

SQL92

There is no SET variable in SQL92. The SQL92 syntax for SET TIME ZONE is slightly different, allowing only a single integer value for time zone specification:

```
SET TIME ZONE { interval_value_expression | LOCAL }
```

SHOW

Name

SHOW Shows run-time parameters for session

SHOW variable

Inputs

variable

Refer to SET for more information on available variables.

Outputs

NOTICE: variable is value SHOW VARIABLE

Message returned if successfully.

NOTICE: Unrecognized variable value

Message returned if value does not exist.

NOTICE: Time zone is unknown SHOW VARIABLE

If the TZ environment variable is not set.

Description

SHOW will display the current configuration parameters for variable during a session.

The session can be configured using SET statement, and values can be restored to the defaults using RESET statement. Parameters and values are case-insensitive.

Notes

The SHOW is a Postgres language extension.

Refer to SET/RESET to set/reset variable values. See also SET TIME ZONE.

Usage

```
-- show DateStyle;
SHOW DateStyle;
NOTICE:DateStyle is Postgres with US (NonEuropean) conventions

-- show Geqo;
SHOW GEQO;
NOTICE:GEQO is ON
```

Compatibility

SQL92

There is no SET defined in SQL92.

UNLISTEN

Name

UNLISTEN Stop listening for notification

```
UNLISTEN { notifyname | * }
```

Inputs

notifyname

Name of previously registered notify condition.

*

All current listen registrations for this backend are cleared.

Outputs

UNLISTEN

Acknowledgement that statement has executed.

Description

UNLISTEN is used to remove an existing NOTIFY registration. UNLISTEN cancels any existing registration of the current Postgres session as a listener on the notify condition notifyname. The special condition wildcard "*" cancels all listener registrations for the current session.

NOTIFY contains a more extensive discussion of the use of LISTEN and NOTIFY.

Notes

classname needs not to be a valid class name but can be any string valid as a name up to 32 characters long.

The backend does not complain if you UNLISTEN something you were not listening for. Each backend will automatically execute UNLISTEN * when exiting.

A restriction in some previous releases of Postgres that a classname which does not correspond to an actual table must be enclosed in double-quotes is no longer present.

Usage

```
postgres=> LISTEN virtual;
LISTEN
postgres=> NOTIFY virtual;
NOTIFY
ASYNC NOTIFY of 'virtual' from backend pid '12317' received

postgres=> UNLISTEN virtual;
UNLISTEN
postgres=> NOTIFY virtual;
NOTIFY
-- notice no NOTIFY event is received
```

```
postgres=>
```

Compatibility

SQL92

There is no UNLISTEN in SQL92.

UPDATE

Name

UPDATE Replaces values of columns in a table

```
UPDATE table SET column = expression [, ...]
    [ FROM fromlist ]
    [ WHERE condition ]
```

Inputs

table

The name of an existing table.

column

The name of a column in table.

expression

A valid expression or value to assign to column.

fromlist

A Postgres non-standard extension to allow columns from other tables to appear in the WHERE condition.

condition

Refer to the SELECT statement for a further description of the WHERE clause.

Outputs

UPDATE #

Message returned if successful. The # means the number of rows updated. If # is equal 0 no rows are updated.

Description

UPDATE changes the values of the columns specified for all rows which satisfy condition. Only the columns to be modified need appear as column.

Array references use the same syntax found in SELECT. That is, either single array elements, a range of array elements or the entire array may be replaced with a single query.

You must have write access to the table in order to modify it, as well as read access to any table whose values are mentioned in the WHERE condition.

Usage

```
--Change word "Drama" with "Dramatic" on column kind:
--
UPDATE films
   SET kind = 'Dramatic'
   WHERE kind = 'Drama';

SELECT * FROM films WHERE kind = 'Dramatic' OR kind = 'Drama';

code |title        |did| date_prod|kind      |len
-----+-------------+---+----------+----------+------
BL101|The Third Man|101|1949-12-23|Dramatic  | 01:44
P_302|Becket       |103|1964-02-03|Dramatic  | 02:28
```

```
M_401|War and Peace|104|1967-02-12|Dramatic  | 05:57
T_601|Yojimbo      |106|1961-06-16|Dramatic  | 01:50
DA101|Das Boot     |110|1981-11-11|Dramatic  | 02:29
```

Compatibility

SQL92

SQL92 defines a different syntax for positioned UPDATE statement:

```
UPDATE table SET column = expression [, ...]
       WHERE CURRENT OF cursor
```

where cursor identifies an open cursor.

VACUUM

Name

VACUUM Clean and analyze a Postgres database

```
VACUUM [ VERBOSE ] [ ANALYZE ] [ table ]
VACUUM [ VERBOSE ] ANALYZE [ table [ (column [, ...] ) ] ]
```

Inputs

VERBOSE

Prints a detailed vacuum activity report for each table.

ANALYZE

Updates column statistics used by the optimizer to determine the most efficient way to execute a query. The statistics represent the disbursion of the data in each column. This information is valuable when several execution paths are possible.

table

The name of a specific table to vacuum. Defaults to all tables.

column

The name of a specific column to analyze. Defaults to all columns.

Outputs

VACUUM

The command has been accepted and the database is being cleaned.

NOTICE: --Relation table--

The report header for table.

NOTICE: Pages 98: Changed 25, Reapped 74, Empty 0, New 0; Tup 1000: Vac 3000, Crash 0, UnUsed 0, MinLen 188, MaxLen 188; Re-using: Free/Avail. Space 586952/586952; EndEmpty/Avail. Pages 0/74. Elapsed 0/0 sec.

The analysis for table itself.

NOTICE: Index index: Pages 28; Tuples 1000: Deleted 3000. Elapsed 0/0 sec.

The analysis for an index on the target table.

Description

VACUUM serves two purposes in Postgres as both a means to reclaim storage and also a means to collect information for the optimizer.

VACUUM opens every class in the database, cleans out records from rolled back transactions, and updates statistics in the system catalogs. The statistics maintained include the number of

tuples and number of pages stored in all classes. Running VACUUM periodically will increase the speed of the database in processing user queries.

Notes

The open database is target for VACUUM.

We recommend that active production databases be cleaned nightly, in order to keep statistics relatively current. The VACUUM query may be executed at any time, however. In particular, after copying a large class into Postgres or after deleting a large number of records, it may be a good idea to issue a VACUUM query. This will update the system catalogs with the results of all recent changes, and allow the Postgres query optimizer to make better choices in planning user queries.

If the server crashes during a VACUUM command, chances are it will leave a lock file hanging around. Attempts to re-run the VACUUM command result in an error message about the creation of a lock file. If you are sure VACUUM is not running, remove the pg_vlock file in your database directory (i.e. PGDATA/base/dbname/pg_vlock).

Usage

The following is an example from running VACUUM on a table in the regression database:

```
regression=> vacuum verbose analyze onek;
NOTICE:  --Relation onek--
NOTICE:  Pages 98: Changed 25, Reapped 74, Empty 0, New 0;
         Tup 1000: Vac 3000, Crash 0, UnUsed 0, MinLen 188, MaxLen 188;
         Re-using: Free/Avail. Space 586952/586952; EndEmpty/Avail.
Pages 0/74.
         Elapsed 0/0 sec.
NOTICE:  Index onek_stringu1: Pages 28; Tuples 1000: Deleted 3000.
Elapsed 0/0 sec.
NOTICE:  Index onek_hundred: Pages 12; Tuples 1000: Deleted 3000.
Elapsed 0/0 sec.
NOTICE:  Index onek_unique2: Pages 19; Tuples 1000: Deleted 3000.
Elapsed 0/0 sec.
NOTICE:  Index onek_unique1: Pages 17; Tuples 1000: Deleted 3000.
Elapsed 0/0 sec.
NOTICE:  Rel onek: Pages: 98 --> 25; Tuple(s) moved: 1000. Elapsed 0/1
sec.
NOTICE:  Index onek_stringu1: Pages 28; Tuples 1000: Deleted 1000.
Elapsed 0/0 sec.
NOTICE:  Index onek_hundred: Pages 12; Tuples 1000: Deleted 1000.
Elapsed 0/0 sec.
NOTICE:  Index onek_unique2: Pages 19; Tuples 1000: Deleted 1000.
Elapsed 0/0 sec.
NOTICE:  Index onek_unique1: Pages 17; Tuples 1000: Deleted 1000.
Elapsed 0/0 sec.
VACUUM
```

Compatibility

SQL92

There is no VACUUM statement in SQL92.

Chapter 17. Utility Applications

This is reference information for the Postgres support utilities.

createdb

Name

createdb Create a new Postgres database

```
createdb [ dbname ]
createdb [ -h host ] [ -p port ]
    [ -D datadir ]
    [ -u ] [ dbname ]
```

Inputs

-h host

Specifies the hostname of the machine on which the postmaster is running. Defaults to using a local Unix domain socket rather than an IP connection..

-p port

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

-u

Use password authentication. Prompts for username and password.

-D datadir

Specifies the alternate database location for this database installation. This is the location of the installation system tables, not the location of this specific database, which may be different.

dbname

Specifies the name of the database to be created. The name must be unique among all Postgres databases in this installation. dbname defaults to the value of the USER environment variable.

Outputs

createdb will create files in the PGDATA/dbname/ data area for the new database.

Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port 'port'? createdb: database creation failed on dbname.

createdb could not attach to the postmaster process on the specified host and port. If you see this message, ensure that the postmaster is running on the proper host and that you have specified the proper port. If your site uses an

```
    authentication system, ensure that you have obtained the
    required authentication credentials.
Connection to database 'template1' failed. FATAL 1: SetUserId:
user 'username' is not in 'pg_shadow' createdb: database
creation failed on dbname.
    You do not have a valid entry in the relation pg_shadow and
    and will not be allowed to access Postgres. Contact your
    Postgres administrator.
ERROR: user 'username' is not allowed to create/destroy
databases createdb: database creation failed on dbname.
    You do not have permission to create new databases. Contact
    your Postgres site administrator.
ERROR: createdb: database 'dbname' already exists. createdb:
database creation failed on dbname.
    The database already exists.
createdb: database creation failed on dbname.
    An internal error occurred in psql or in the backend server.
    Ensure that your site administrator has properly installed
    Postgresand initialized the site with initdb.
```

Note: createdb internally runs CREATE DATABASE from psql while connected to the template1 database.

Description

createdb creates a new Postgres database. The person who executes this command becomes the database administrator, or DBA, for this database and is the only person, other than the Postgres super-user, who can destroy it.

createdb is a shell script that invokes psql. Hence, a postmaster process must be running on the database server host before createdb is executed. The PGOPTION and PGREALM environment variables will be passed on to psql and processed as described in *psql*.

Usage

To create the database demo using the postmaster on the local host, port 5432:

```
createdb demo
```

To create the database demo using the postmaster on host eden, port 5000:

```
createdb -p 5000 -h eden demo
```

createuser

Name

createuser Create a new Postgres user

```
createuser [ username ]
createuser [ -h host ] [ -p port ]
    [ -i userid ]
    [ -d | -D ] [ -u | -U ] [ username ]
```

Inputs

-h host

Specifies the hostname of the machine on which the postmaster is running. Defaults to using a local Unix domain socket rather than an IP connection..

-p port

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

-d

Allows the user to create databases.

-D

Forbids the user to create databases.

-i userid

Specifies the numeric identifier to be associated with this user. This identifier must be unique among all Postgres users, and is not required to match the operating system UID. You will be prompted for an identifier if none is specified on the command line, and it will suggest an identifier matching the UID.

-u

Allows the user to create other users.

-U

Forbids the user to create other users.

username

Specifies the name of the Postgres user to be created. This name must be unique among all Postgres users. You will be prompted for a name if none is specified on the command line.

Outputs

createuser will add an entry in the pg_user or pg_shadow system table.

Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port 'port'? createuser: database access failed.

createuser could not attach to the postmaster process on the specified host and port. If you see this message, ensure that the postmaster is running on the proper host and that

```
    you have specified the proper port. If your site uses an
    authentication system, ensure that you have obtained the
    required authentication credentials.
Connection to database 'template1' failed. FATAL 1: SetUserId:
user 'username' is not in 'pg_shadow' createuser: database
access failed.
    You do not have a valid entry in the relation pg_shadow and
    and will not be allowed to access Postgres. Contact your
    Postgres administrator.
createuser: username cannot create users.
    You do not have permission to create new users; contact your
    Postgres site administrator.
createuser: user "username" already exists
    The user to be added already has an entry in the pg_shadow
    class.
database access failed
    An internal error occurred in psql or in the backend server.
    Ensure that your site administrator has properly installed
    Postgresand initialized the site with initdb.
```

Note: createuser internally runs CREATE USER from psql while connected to the template1 database.

Description

createuser creates a new Postgres user. Only users with usesuper set in the pg_shadow class can create new Postgres users. As shipped, the user postgres can create users.

createuser is a shell script that invokes psql. Hence, a postmaster process must be running on the database server host before createuser is executed. The PGOPTION and PGREALM environment variables will be passed on to psql and processed as described in *psql*. Once invoked, createuser will ask a series of questions to obtain parameters not specified on the command line. The new user's database login name and a numeric user identifier must be specified.

Note: The Postgres user identifier does not need to be the same as the user's Unix UID. However, typically they are assigned to be the same.

You must also describe the privileges of the new user for security purposes. Specifically, you will be asked whether the new user should be able to act as Postgres super-user, whether the new user may create new databases and whether the new user is allowed to create other new users.

destroydb

Name

destroydb Create a new Postgres database

```
destroydb [ dbname ]
destroydb [ -h host ] [ -p port ]
    [ -i ] [ dbname ]
```

Inputs

-h host

Specifies the hostname of the machine on which the postmaster is running. Defaults to using a local Unix domain socket rather than an IP connection..

-p port

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

-i

Run in interactive mode. Prompts for confirmation before destroying a database.

dbname

Specifies the name of the database to be destroyed. The database must be one of the existing Postgres databases in this installation. dbname defaults to the value of the USER environment variable.

Outputs

destroydb will remove files from the PGDATA/dbname/ data area for the existing database.

Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port 'port'? destroydb: database destroy failed on dbname.

destroydb could not attach to the postmaster process on the specified host and port. If you see this message, ensure that the postmaster is running on the proper host and that you have specified the proper port. If your site uses an

```
    authentication system, ensure that you have obtained the
    required authentication credentials.
Connection to database 'template1' failed. FATAL 1: SetUserId:
user 'username' is not in 'pg_shadow' destroydb: database
destroy failed on dbname.
    You do not have a valid entry in the relation pg_shadow and
    and will not be allowed to access Postgres. Contact your
    Postgres administrator.
ERROR: user 'username' is not allowed to create/destroy
databases destroydb: database destroy failed on dbname.
    You do not have permission to destroy (or create) databases.
    Contact your Postgres site administrator.
ERROR: destroydb: database 'dbname' does not exist. destroydb:
database destroy failed on dbname.
    The database to be removed does not have an entry in the
    pg_database class.
ERROR: destroydb: database 'dbname' is not owned by you.
destroydb: database destroy failed on dbname.
    You are not the Database Administrator (DBA) for the
    specified database.
destroydb: database destroy failed on dbname.
    An internal error occurred in psql or in the backend server.
    Ensure that your site administrator has properly installed
    Postgresand initialized the site with initdb.
```

Note: destroydb internally runs DESTROY DATABASE from psql while connected to the template1 database.

Description

destroydb destroys an existing Postgres database. The person who executes this command must be the database administrator, or DBA, or must be the Postgres super-user. The program runs silently; no confirmation message will be displayed. After the database is destroyed, a Unix shell prompt will reappear.

All references to the database are removed, including the directory containing this database and its associated files.

destroydb is a shell script that invokes psql. Hence, a postmaster process must be running on the database server host before destroydb is executed. The PGOPTION and PGREALM environment variables will be passed on to psql and processed as described in *psql*.

Usage

To destroy the database demo using the postmaster on the local host, port 5432:

```
destroydb demo
```

To destroy the database demo using the postmaster on host eden, port 5000:

```
destroydb -p 5000 -h eden demo
```

destroyuser

Name

destroyuser Destroy a Postgres user and associated databases

```
destroyuser [ username ]
destroyuser [ -h host ] [ -p port ]
    [ username ]
```

Inputs

-h host

Specifies the hostname of the machine on which the postmaster is running. Defaults to using a local Unix domain socket rather than an IP connection..

-p port

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

username

Specifies the name of the Postgres user to be removed. This name must exist in the Postgres installation. You will be prompted for a name if none is specified on the command line.

Outputs

destroyuser will remove an entry in the pg_user or pg_shadow system table, and will remove all databases for which that user is the administrator (DBA).

Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port 'port'? destroyuser: database access failed.

destroyuser could not attach to the postmaster process on the specified host and port. If you see this message, ensure that the postmaster is running on the proper host and that you have specified the proper port. If your site uses an

```
    authentication system, ensure that you have obtained the
    required authentication credentials.
Connection to database 'template1' failed. FATAL 1: SetUserId:
user 'username' is not in 'pg_shadow' destroyuser: database
access failed.
    You do not have a valid entry in the relation pg_shadow and
    and will not be allowed to access Postgres. Contact your
    Postgres administrator.
destroyuser: username cannot delete users.
    You do not have permission to delete users; contact your
    Postgres site administrator.
destroyuser: user "username" already exists
    The user to be added already has an entry in the pg_shadow
    class.
database access failed
    An internal error occurred in psql or in the backend server.
    Ensure that your site administrator has properly installed
    Postgresand initialized the site with initdb.
destroydb on dbname failed - exiting
    An internal error occurred in psql or in the backend server.
    There was possibly a Unix permissions problem with the
    specified database.
delete of user username was UNSUCCESSFUL
    An internal error occurred in psql or in the backend server.
```

Note: destroyuser internally runs DROP USER from psql while connected to the template1 database.

Description

destroyuser removes an existing Postgres user and the databases for which that user is database administrator. Only users with usesuper set in the pg_shadow class can destroy Postgres users. As shipped, the user postgres can remove users.

destroyuser is a shell script that invokes psql. Hence, a postmaster process must be running on the database server host before destroyuser is executed. The PGOPTION and PGREALM environment variables will be passed on to psql and processed as described in *psql*.

Once invoked, destroyuser will warn you about the databases that will be destroyed in the process and permit you to abort the removal of the user if desired.

initdb

Name

initdb Create a new Postgres database installation

```
initdb [ --pgdata=dbdir | -r dbdir ]
    [ --pglib=libdir | -l libdir ]
    [ --template=template | -t template ]
    [ --username=name | -u name ]
    [ --noclean | -n ] [ --debug | -d ]
```

Inputs

--pglib=libdir

-l libdir

PGLIB

Where are the files that make up Postgres? Apart from files that have to go in particular directories because of their function, the files that make up the Postgres software were installed in a directory called the libdir directory. An example of a file that will be found there that initdb needs is global1.bki.source, which contains all the information that goes into the shared catalog tables.

--pgdata=dbdir

-r dbdir

PGDATA

Where in your Unix filesystem do you want the database data to go? The top level directory is called the PGDATA directory.

--username=name

-u name

PGUSER

Who will be the Postgres superuser for this database system? The Postgres superuser is a Unix user who owns all files that store the database system and also owns the postmaster and backend processes that access them. Or just let it default to you (the Unix user who runs initdb).

Note: Only the Unix superuser (root) can create a database system with an owner different from the Postgres superuser.

Other, less commonly used, parameters are also available:

--template=template

-t template

Replace the template1 database in an existing database system, and don’t touch anything else. This is useful when you need to upgrade your template1 database using initdb from a newer release of Postgres, or when your template1 database has become corrupted by some system problem. Normally the contents of template1 remain constant throughout the life of the database system. You can’t

destroy anything by running initdb with the --template option.

--noclean

-n

By default, when initdb determines that error prevent it from completely creating the database system, it removes any files it may have created before determining that it can't finish the job. That includes any core files left by the programs it invokes. This option inhibits any tidying-up and is thus useful for debugging.

--debug

-d

Print debugging output from the bootstrap backend. The bootstrap backend is the program initdb uses to create the catalog tables. This option generates a tremendous amount of output. It also turns off the final vacuuming step.

Files are also input to initdb:

postconfig

If appearing somewhere in the Unix command search path (defined by the PATH environment variable). This is a program that specifies defaults for some of the command options. See below.

PGLIB/global1.bki.source

Contents for the shared catalog tables in the new database system. This file is part of the Postgres software.

PGLIB/local1_template1.bki.source

Contents for the template1 tables in the new database system. This file is part of the Postgres software.

Outputs

initdb will create files in the PGDATA data area which are the system tables and framework for a complete installation.

Description

initdb creates a new Postgres database system. A database system is a collection of databases that are all administered by the same Unix user and managed by a single postmaster.

Creating a database system consists of creating the directories in which the database data will live, generating the shared catalog tables (tables that don't belong to any particular database), and creating the template1 database. What is the template1 database? When you create a database, Postgres does it by copying everything from the template1 database. It contains catalog tables filled in for things like the builtin types.

After initdb creates the database, it completes the initialization by running vacuum, which resets some optimization parameters.

There are three ways to give parameters to initdb. First, you can use initdb command options. Second, you can set environment variables before invoking initdb. Third, you can have a program called postconfig in your Unix command search path. initdb invokes that program and

that program then writes initdb parameters to its standard output stream. This third option is not a common thing to do, however.

Command options always override parameters specified any other way. The values returned by postconfig override any environment variables, but your postconfig program may base its output on the environment variables if you want their values to be used.

The value that postconfig outputs must have the format

```
var1=value1 var2=value2 ...
```

It can output nothing if it doesn't want to supply any parameters. The var values are equal to the corresponding environment variable names. For example,

```
PGDATA=/tmp/postgres_test
```

has the same effect as invoking initdb with an environment variable called PGDATA whose value is /tmp/postgres_test.

initlocation

Name

initlocation Create a secondary Postgres database storage area

```
initlocation [ --location=altdir | -D altdir ]
    [ --username=name | -u name ]
    [ altdir ]
```

Inputs

--location=altdir

-D altdir

altdir

Where in your Unix filesystem do you want alternate databases to go? The top level directory is called the PGDATA directory, so you might want to point your first alternate location at PGDATA2.

--username=name

-u name

PGUSER

Who will be the Unix filesystem owner of this database storage area? The Postgres superuser is a Unix user who owns all files that store the database system and also owns the postmaster and backend processes that access them. Usually, this is the user who should run initlocation and who will thus have ownership of the directories and files.

Note: Only the Unix superuser can create a database system with a different user as the Postgres superuser. Specifying a user other than the Postgres superuser may lead to database security and data integrity problems. Refer to the PostgreSQL Administrator's Guide for more information.

Outputs

initlocation will create directories in the specified place.

We are initializing the database area with username postgres (uid=500). This user will own all the files and must also own the server process. Creating Postgres database system directory altdir Creating Postgres database system directory altdir

Successful completion.

We are initializing the database area with username postgres (uid=500). This user will own all the files and must also own the server process. Creating Postgres database system directory /usr/local/src/testlocation mkdir: cannot make directory 'altdir': Permission denied

You do not have filesystem permission to write to the specified directory area.

Valid username not given. You must specify the username for the Postgres superuser for the database system you are initializing,

```
either with the --username option or by default to the USER
environment variable.
    The username which you have specified is not the Postgres
    superuser.
Can't tell what username to use. You don't have the USER
environment variable set to your username and didn't specify the
--username option
    Specify the --username command line option.
```

Description

initlocation creates a new Postgres secondary database storage area. A secondary storage area contains a required tree of directories with the correct file permissions on those directories.

Creating a database storage area consists of creating the directories in which database data might live.

There are two kinds of arguments for initlocation. First, you can specify an environment variable (e.g. PGDATA2). This environment variable should be known to the backend for later use in CREATE DATABASE/WITH LOCATION or createdb -D altdir. However, the backend daemon must have this variable in it's environment for this to succeed. Second, you may be able to specify an explicit absolute path to the top directory of the storage area. However,this second option is possible only if explicitly enabled during the Postgres installation. It is usually disabled to alleviate security and data integrity concerns.

> Note: Postgres will add /base/ to the specified path to create the storage area.
>
> The backend requires that any argument to WITH LOCATION which is in all uppercase and which has no path delimiters is an environment variable.

Usage

To create a database in an alternate location, using an environment variable:

```
% setenv PGDATA2 /opt/postgres/data

% initlocation PGDATA2
% createdb -D PGDATA2
```

pg_dump

Name

pg_dump Extract a Postgres database into a script file

```
pg_dump [ dbname ]
pg_dump [ -h host ] [ -p port ]
    [ -t table ]
    [ -f outputfile ]
    [ -a ] [ -d ] [ -D ] [ -o ] [ -s ] [ -u ] [ -v ] [ -z ]
    [ dbname ]
```

Inputs

pg_dump accepts the following command line arguments:

dbname

Specifies the name of the database to be extracted. dbname defaults to the value of the USER environment variable.

-a

Dump out only the data, no schema (definitions).

-d

Dump data as proper insert strings.

-D

Dump data as inserts with attribute names

-f filename

Specifies the output file. Defaults to stdout.

-n

Suppress double quotes around identifiers unless absolutely necessary. This may cause trouble loading this dumped data if there are reserved words used for identifiers.

-o

Dump object identifiers (OIDs) for every table.

-s

Dump out only the schema (definitions), no data.

-t table

Dump data for table only.

-u

Use password authentication. Prompts for username and password.

-v

Specifies verbose mode

-z

Include ACLs (grant/revoke commands) and table ownership information.

pg_dump also accepts the following command line arguments for connection parameters:

-h host

Specifies the hostname of the machine on which the postmaster is running. Defaults to using a local Unix domain socket rather than an IP connection..

-p port

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

-u

Use password authentication. Prompts for username and password.

Outputs

pg_dump will create a file or write to stdout.

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

pg_dump could not attach to the postmaster process on the specified host and port. If you see this message, ensure that the postmaster is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

Connection to database 'dbname' failed. FATAL 1: SetUserId: user 'username' is not in 'pg_shadow'

You do not have a valid entry in the relation pg_shadow and and will not be allowed to access Postgres. Contact your Postgres administrator.

dumpSequence(table): SELECT failed

You do not have permission to read the database. Contact your Postgres site administrator.

Note: pg_dump internally executes SELECT statements. If you have problems running pg_dump, make sure you are able to select information from the database using, for example, psql.

Description

pg_dump is a utility for dumping out a Postgres database into a script file containing query commands. The script files are in text format and can be used to reconstruct the database, even on other machines and other architectures. pg_dump will produce the queries necessary to re-generate all user-defined types, functions, tables, indices, aggregates, and operators. In

addition, all the data is copied out in text format so that it can be readily copied in again, as well as imported into tools for editing.

pg_dump is useful for dumping out the contents of a database to move from one Postgres installation to another. After running pg_dump, one should examine the output script file for any warnings, especially in light of the limitations listed below.

Notes

pg_dump has a few limitations. The limitations mostly stem from difficulty in extracting certain meta-information from the system catalogs.

partial indices

pg_dump does not understand partial indices. The reason is the same as above; partial index predicates are stored as plans.

large objects

pg_dump does not handle large objects. Large objects are ignored and must be dealt with manually.

Usage

To dump a database of the same name as the user:

```
% pg_dump > db.out
```

To reload this database:

```
psql -e database < db.out
```

pg_dumpall

Name

pg_dumpall Extract all Postgres databases into a script file

```
pg_dumpall
pg_dumpall [ -h host ] [ -p port ]
    [ -a ] [ -d ] [ -D ] [ -o ] [ -s ] [ -u ] [ -v ] [ -z ]
```

Inputs

pg_dumpall accepts the following command line arguments:

-a

Dump out only the data, no schema (definitions).

-d

Dump data as proper insert strings.

-D

Dump data as inserts with attribute names

-n

Suppress double quotes around identifiers unless absolutely necessary. This may cause trouble loading this dumped data if there are reserved words used for identifiers.

-o

Dump object identifiers (OIDs) for every table.

-s

Dump out only the schema (definitions), no data.

-u

Use password authentication. Prompts for username and password.

-v

Specifies verbose mode

-z

Include ACLs (grant/revoke commands) and table ownership information.

pg_dumpall also accepts the following command line arguments for connection parameters:

-h host

Specifies the hostname of the machine on which the postmaster is running. Defaults to using a local Unix domain socket rather than an IP connection..

-p port

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening

```
   for connections. The port number defaults to 5432, or the
   value of the PGPORT environment variable (if set).

-u
   Use password authentication. Prompts for username and
   password.
```

Outputs

```
pg_dumpall will create a file or write to stdout.

Connection to database 'template1' failed. connectDB() failed:
Is the postmaster running and accepting connections at 'UNIX
Socket' on port 'port'?
   pg_dumpall could not attach to the postmaster process on the
   specified host and port. If you see this message, ensure
   that the postmaster is running on the proper host and that
   you have specified the proper port. If your site uses an
   authentication system, ensure that you have obtained the
   required authentication credentials.

Connection to database 'dbname' failed. FATAL 1: SetUserId: user
'username' is not in 'pg_shadow'
   You do not have a valid entry in the relation pg_shadow and
   and will not be allowed to access Postgres. Contact your
   Postgres administrator.

dumpSequence(table): SELECT failed
   You do not have permission to read the database. Contact
   your Postgres site administrator.
```

Note: pg_dumpall internally executes SELECT statements. If you have problems running pg_dumpall, make sure you are able to select information from the database using, for example, psql.

Description

pg_dumpall is a utility for dumping out all Postgres databases into one file. It also dumps the pg_shadow table, which is global to all databases. pg_dumpall includes in this file the proper commands to automatically create each dumped database before loading.

pg_dumpall takes all pg_dump options, but -f, -t and dbname should be omitted.

Refer to *pg_dump* for more information on this capability.

Usage

To dump all databases:

```
% pg_dumpall -o > db.out
```

Tip: You can use most pg_dump options for pg_dumpall.

To reload this database:

```
psql -e template1 < db.out
```

Tip: You can use most psql options when reloading.

psql

Name

psql Postgres interactive client

```
psql [ dbname ]
psql -A [ -c query ] [ -d dbname ]
    -e [ -f filename ] [ -F separator ] [ -h hostname ]
    -Hln [ -o filename ] [ -p port ] -qsSt ]
    [ -T table_options ] -ux [ dbname ]
```

Inputs

psql accepts many command-line arguments, a rich set of meta-commands, and the full SQL language supported by Postgres. The most common command-line arguments are:

dbname

The name of an existing database to access. dbname defaults to the value of the USER environment variable or, if that's not set, to the Unix account name of the current user.

-c query

A single query to run. psql will exit on completion.

The full set of command-line arguments and meta-commands are described in a subsequent section.

There are some environment variables which can be used in liu of command line arguments. Additionally, the Postgres frontend library used by the psql application looks for other optional environment variables to configure, for example, the style of date/time representation and the local time zone. Refer to the chapter on libpq in the Programmer's Guide for more details.

Outputs

psql returns 0 to the shell on successful completion of all queries, 1 for errors, 2 for abrupt disconnection from the backend. The default TAB delimiter is used. psql will also return 1 if the connection to a database could not be made for any reason.

Description

psql is a character-based front-end to Postgres. It enables you to type in queries interactively, issue them to Postgres, and see the query results.

psql is a Postgres client application. Hence, a postmaster process must be running on the database server host before psql is executed. In addition, the correct parameters to identify the database server, such as the postmaster host name, may need to be specified as described below.

When psql starts, it reads SQL commands from /etc/psqlrc and then from $(HOME)/.psqlrc. This allows SQL commands like SET which can be used to set the date style to be run at the start of every session.

Connecting To A Database

psql attempts to make a connection to the database at the hostname and port number specified on the command line. If the connection could not be made for any reason (e.g. insufficient privileges, postmaster is not running on the server, etc) .IR psql will return an error that says

```
Connection to database failed.
```

The reason for the connection failure is not provided.

Entering Queries

In normal operation, psql provides a prompt with the name of the database that psql is current connected to followed by the string "=>". For example,

```
$ psql testdb
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL
   type \e? for help on slash commands
   type \eq to quit
   type \eg or terminate with semicolon to execute query
 You are currently connected to the database: testdb
testdb=>
```

At the prompt, the user may type in SQL queries. Unless the -S option is set, input lines are sent to the backend when a query-terminating semicolon is reached.

Whenever a query is executed, psql also polls for asynchronous notification events generated by LISTEN and NOTIFY.

psql can be used in a pipe sequence, and automatically detects when it is not listening or talking to a real tty.

Command-line Options

psql understands the following command-line options:

-A

Turn off fill justification when printing out table elements.

-c query

Specifies that psql is to execute one query string, query, and then exit. This is useful for shell scripts, typically in conjunction with the -q option in shell scripts.

-d dbname

Specifies the name of the database to connect to. This is equivalent to specifying dbname as the last field in the command line.

-e

Echo the query sent to the backend

-f filename

Use the file filename as the source of queries instead of reading queries interactively. This file must be specified for and visible to the client frontend.

-F separator

Use separator as the field separator. The default is an ASCII vertical bar ("|").

-h hostname

Specifies the host name of the machine on which the postmaster is running. Without this option, communication is performed using local Unix domain sockets.

-H

Turns on HTML 3.0 tabular output.

-l

Lists all available databases, then exit. Other non-connection options are ignored.

-n

Do not use the readline library for input line editing and command history.

-o filename

Put all output into file filename. The path must be writable by the client.

-p port

Specifies the TCP/IP port or, by omission, the local Unix domain socket file extension on which the postmaster is listening for connections. Defaults to the value of the PGPORT environment variable, if set, or to 5432.

-q

Specifies that psql should do its work quietly. By default, it prints welcome and exit messages and prompts for each query, and prints out the number of rows returned from a query. If this option is used, none of this happens. This is useful with the -c option.

-s

Run in single-step mode where the user is prompted for each query before it is sent to the backend.

-S

Runs in single-line mode where each query is terminated by a newline, instead of a semicolon.

-t

Turn off printing of column names. This is useful with the -c option in shell scripts.

-T table_options

Allows you to specify options to be placed within the table ... tag for HTML 3.0 tabular output.For example, border will give you tables with borders. This must be used in conjunction with the -H option.

-u

Asks the user for the user name and password before connecting to the database. If the database does not require password authentication then these are ignored. If the option is not used (and the PGPASSWORD environment variable is not set) and the database requires password authentication, then the connection will fail. The user name is ignored anyway.

-x

Turns on extended row format mode. When enabled each row will have its column names printed on the left with the column values printed on the right. This is useful for rows which are otherwise too long to fit into one screen line. HTML row output supports this mode also.

You may set environment variables to avoid typing some of the above options. See the section on environment variables below.

psql Meta-Commands

Anything you enter in psql that begins with an unquoted backslash is a psql meta-command. Anything else is SQL and simply goes into the current query buffer (and once you have at least one complete query, it gets automatically submitted to the backend). psql meta-commands are also called slash commands.

The format of a psql command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of white space characters.

With single character command verbs, you don't actually need to separate the command verb from the argument with white space, for historical reasons. You should anyway.

The following meta-commands are defined:

`\a`

Toggle field alignment when printing out table elements.

`\C caption`

Set the HTML3.0 table caption to caption .

\connect dbname [username]

Establish a connection to a new database, using the default username if none is specified. The previous connection is closed.

`\copy dbname { FROM | TO } filename`

Perform a frontend (client) copy. This is an operation that runs a SQL COPY command, but instead of the backend reading or writing the specified file, and consequently requiring

backend access and special user privilege, psql reads or writes the file and routes the data to or from the backend. The default tab delimiter is used.

Tip: This operation is not as efficient as the SQL COPY command because all data must pass through the client/server IP or socket connection. For large amounts of data this other technique may be preferable.

\d [table]

List tables in the database, or if table is specified, list the columns in that table. If table name is specified as an asterisk (*), list all tables and column information for each tables.

`\da`

List all available aggregates.

`\dd object`

List the description from pg_description of the specified object, which can be a table, table.column, type, operator, or aggregate.

Tip: Not all objects have a description in pg_description. This meta-command can be useful to get a quick description of a native Postgres feature.

`\df`

List functions.

`\di`

List only indexes.

`\do`

List only operators.

`\ds`

List only sequences.

`\dS`

List system tables and indexes.

`\dt`

List only non-system tables.

`\dT`

List types.

\e [filename]

Edit the current query buffer or the contents of the file filename.

\E [filename]

Edit the current query buffer or the contents of the file filename and execute it upon editor exit.

\f [separator]

Set the field separator. Default is a single blank space.

\g [{ filename | |command }]

Send the current query input buffer to the backend and optionally save the output in filename or pipe the output into a separate Unix shell to execute command.

\h [command]

Give syntax help on the specified SQL command. If command is not a defined SQL command (or is not documented in psql), or if command is not specified, then psql will list

all the commands for which syntax help is available. If command is an asterisk (*), then give syntax help on all SQL commands.

`\H`

Toggle HTML3 output. This is equivalent to the -H command-line option.

`\i filename`

Read queries from the file filename into the query input buffer.

`\l`

List all the databases in the server.

`\m`

Toggle the old monitor-like table display, which includes border characters surrounding the table. This is standard SQL output. By default, psql includes only field separators between columns.

\o [{ filename | |command }]

Save future query results to the file filename or pipe future results into a separate Unix shell to execute command. If no arguments are specified, send query results to stdout.

`\p`

Print the current query buffer.

`\q`

Quit the psql program.

`\r`

Reset(clear) the query buffer.

\s [filename]

Print or save the command line history to filename. If filename is omitted, do not save subsequent commands to a history file. This option is only available if psql is configured to use readline.

`\t`

Toggle display of output column name headings and row count footer (defaults to on).

`\T table_options`

Allows you to specify options to be placed within the table ... tag for HTML 3.0 tabular output.For example, border will give you tables with borders. This must be used in conjunction with the \H meta-command.

`\x`

Toggles extended row format mode. When enabled each row will have its column names printed on the left with the column values printed on the right. This is useful for rows

which are otherwise too long to fit into one screen line. HTML row output mode supports this flag too.

`\w filename`

Outputs the current query buffer to the file filename.

`\z`

Produces a list of all tables in the database with their appropriate ACLs (grant/revoke permissions) listed.

\! [command]

Escape to a separate Unix shell or execute the Unix command command.

`\?`

Get help information about the slash (\ ) commands.

SQL References

Selected references and readings for SQL and Postgres.

SQL Reference Books

The Practical SQL Handbook, Judity Bowman, Sandra Emerson, and Marcy Damovsky, ISBN: 0-201-44787-8, 1996, Addison-Wesley.

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

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

PostgreSQL-Specific Documentation

The PostgreSQL Administrator's Guide, 1998-10-01, The PostgreSQL Global Development Group.

The PostgreSQL Developer's Guide, 1998-10-01, The PostgreSQL Global Development Group.

The PostgreSQL Programmer's Guide, 1998-10-01, The PostgreSQL Global Development Group.

The PostgreSQL Tutorial Introduction, 1998-10-01, The PostgreSQL Global Development Group.

The PostgreSQL User's Guide, 1998-10-01, The PostgreSQL Global Development Group.

The Postgres95 User Manual, A. Yu and J. Chen, The POSTGRES Group, Sept. 5, 1995, University of California, Berkeley CA.

Proceedings and Articles

Partial indexing in POSTGRES : research project, Nels Olson, 1993, ISSN: UCB Engin T7.49.1993 O676, University of California, Berkeley CA.

A Unified Framework for Version Modeling Using Production Rules in a Database System, L. Ong and J. Goh, April, 1990, ISSN: ERL Technical Memorandum M90/33, University of California, Berkeley CA.

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

Generalized partial indexes, P. Seshadri and A. Swami, 1995, ISSN: Cat. No.95CH35724, IEEE Computer Society Press.

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 case for partial indexes (DBMS), M. Stonebraker.

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.