PostgreSQL 7.3.2 Reference Manual

The PostgreSQL Global Development Group

PostgreSQL 7.3.2 Reference Manual

by The PostgreSQL Global Development Group

Copyright © 1996-2002 by The PostgreSQL Global Development Group

Legal Notice

PostgreSQL is Copyright © 1996-2002 by the PostgreSQL Global Development Group and is distributed under the terms of the license of the University of California below.

Postgres95 is Copyright © 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 MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Table of Contents

Preface

The entries in this *Reference Manual* are meant to provide in reasonable length an authoritative, complete, and formal summary about their respective subjects. More information about the use of PostgreSQL, in narrative, tutorial, or example form, may be found in other parts of the PostgreSQL documentation set. See the cross-references listed on each reference page.

The *Reference Manual* entries are also available as traditional “man” pages.

I. SQL Commands

This part contains reference information for the SQL commands supported by PostgreSQL. By “SQL” the language in general is meant; information about the standards conformance and compatibility of each command can be found on the respective reference page.

ABORT

Name

`ABORT` — abort the current transaction

Synopsis

```
ABORT [ WORK | TRANSACTION ]
```

Inputs

None.

Outputs

`ROLLBACK`

Message returned if successful.

`WARNING: ROLLBACK: no transaction in progress`

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 `COMMIT` to successfully terminate a transaction.

Usage

To abort all changes:

```
ABORT WORK;
```

Compatibility

SQL92

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

ALTER DATABASE

Name

`ALTER DATABASE` — change a database

Synopsis

```
ALTER DATABASE name SET variable { TO | = } { value | DEFAULT }
ALTER DATABASE name RESET variable
```

Description

`ALTER DATABASE` is used to change the session default of a run-time configuration variable for a PostgreSQL database. Whenever a new session is subsequently started in that database, the specified value becomes the session default value. The database-specific default overrides whatever setting is present in `postgresql.conf` or has been received from the postmaster.

Only a superuser or the database owner can change the session defaults for a database.

Parameters

`name`

The name of the database whose session defaults are to be altered.

`variable`
`value`

Set the session default for this database of the specified configuration variable to the given value. If *`value`* is `DEFAULT` or, equivalently, `RESET` is used, the database-specific variable setting is removed and the system-wide default setting will be inherited in new sessions. Use `RESET ALL` to clear all settings.

See *SET* and the *Administrator's Guide* for more information about allowed variable names and values.

Diagnostics

`ALTER DATABASE`

Message returned if the alteration was successful.

`ERROR: database "dbname" does not exist`

Error message returned if the specified database is not known to the system.

Notes

Using *ALTER USER*, it is also possible to tie a session default to a specific user rather than a database. User-specific settings override database-specific ones if there is a conflict.

Examples

To disable index scans by default in the database `test`:

```
ALTER DATABASE test SET enable_indexscan TO off;
```

Compatibility

The `ALTER DATABASE` statement is a PostgreSQL extension.

See Also

ALTER USER, *CREATE DATABASE*, *DROP DATABASE*, *SET*

ALTER GROUP

Name

`ALTER GROUP` — add users to a group or remove users from a group

Synopsis

```
ALTER GROUP name ADD USER username [, ... ]
ALTER GROUP name DROP USER username [, ... ]
```

Inputs

`name`

The name of the group to modify.

`username`

Users which are to be added or removed from the group. The user names must exist.

Outputs

`ALTER GROUP`

Message returned if the alteration was successful.

Description

`ALTER GROUP` is used to add or remove users from a group. Only database superusers can use this command. Adding a user to a group does not create the user. Similarly, removing a user from a group does not drop the user itself.

Use *CREATE GROUP* to create a new group and *DROP GROUP* to remove a group.

Usage

Add users to a group:

```
ALTER GROUP staff ADD USER karl, john;
```

Remove a user from a group:

```
ALTER GROUP workers DROP USER beth;
```

Compatibility

SQL92

There is no `ALTER GROUP` statement in SQL92. The concept of roles is similar.

ALTER TABLE

Name

`ALTER TABLE` — change the definition of a table

Synopsis

```
ALTER TABLE [ ONLY ] table [ * ]
    ADD [ COLUMN ] column type [ column_constraint [ ... ] ]
ALTER TABLE [ ONLY ] table [ * ]
    DROP [ COLUMN ] column [ RESTRICT | CASCADE ]
ALTER TABLE [ ONLY ] table [ * ]
    ALTER [ COLUMN ] column { SET DEFAULT value | DROP DEFAULT }
ALTER TABLE [ ONLY ] table [ * ]
    ALTER [ COLUMN ] column { SET | DROP } NOT NULL
ALTER TABLE [ ONLY ] table [ * ]
    ALTER [ COLUMN ] column SET STATISTICS integer
ALTER TABLE [ ONLY ] table [ * ]
    ALTER [ COLUMN ] column SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
ALTER TABLE [ ONLY ] table [ * ]
    RENAME [ COLUMN ] column TO new_column
ALTER TABLE table
    RENAME TO new_table
ALTER TABLE [ ONLY ] table [ * ]
    ADD table_constraint
ALTER TABLE [ ONLY ] table [ * ]
    DROP CONSTRAINT constraint_name [ RESTRICT | CASCADE ]
ALTER TABLE table
    OWNER TO new_owner
```

Inputs

table

The name (possibly schema-qualified) of an existing table to alter. If `ONLY` is specified, only that table is altered. If `ONLY` is not specified, the table and all its descendant tables (if any) are updated. `*` can be appended to the table name to indicate that descendant tables are to be scanned, but in the current version, this is the default behavior. (In releases before 7.1, `ONLY` was the default behavior.) The default can be altered by changing the `SQL_INHERITANCE` configuration option.

column

Name of a new or existing column.

type

Type of the new column.

new_column

New name for an existing column.

new_table

New name for the table.

`table_constraint`

New table constraint for the table.

`constraint_name`

Name of an existing constraint to drop.

`new_owner`

The user name of the new owner of the table.

CASCADE

Automatically drop objects that depend on the dropped column or constraint (for example, views referencing the column).

RESTRICT

Refuse to drop the column or constraint if there are any dependent objects. This is the default behavior.

Outputs

`ALTER TABLE`

Message returned from column or table renaming.

`ERROR`

Message returned if table or column is not available.

Description

`ALTER TABLE` changes the definition of an existing table. There are several sub-forms:

ADD COLUMN

This form adds a new column to the table using the same syntax as *CREATE TABLE*.

DROP COLUMN

This form drops a column from a table. Note that indexes and table constraints involving the column will be automatically dropped as well. You will need to say `CASCADE` if anything outside the table depends on the column --- for example, foreign key references, views, etc.

SET/DROP DEFAULT

These forms set or remove the default value for a column. Note that defaults only apply to subsequent `INSERT` commands; they do not cause rows already in the table to change. Defaults may also be created for views, in which case they are inserted into `INSERT` statements on the view before the view's ON INSERT rule is applied.

SET/DROP NOT NULL

  These forms change whether a column is marked to allow NULL values or to reject NULL values. You may only `SET NOT NULL` when the table contains no null values in the column.

SET STATISTICS

  This form sets the per-column statistics-gathering target for subsequent *ANALYZE* operations. The target can be set in the range 0 to 1000; alternatively, set it to -1 to revert to using the system default statistics target.

SET STORAGE

  This form sets the storage mode for a column. This controls whether this column is held inline or in a supplementary table, and whether the data should be compressed or not. `PLAIN` must be used for fixed-length values such as `INTEGER` and is inline, uncompressed. `MAIN` is for inline, compressible data. `EXTERNAL` is for external, uncompressed data and `EXTENDED` is for external, compressed data. `EXTENDED` is the default for all data types that support it. The use of `EXTERNAL` will make substring operations on a TEXT column faster, at the penalty of increased storage space.

RENAME

  The `RENAME` forms change the name of a table (or an index, sequence, or view) or the name of an individual column in a table. There is no effect on the stored data.

ADD *`table_constraint`*

  This form adds a new constraint to a table using the same syntax as *CREATE TABLE*.

DROP CONSTRAINT

  This form drops constraints on a table. Currently, constraints on tables are not required to have unique names, so there may be more than one constraint matching the specified name. All such constraints will be dropped.

OWNER

  This form changes the owner of the table, index, sequence or view to the specified user.

You must own the table to use `ALTER TABLE`; except for `ALTER TABLE OWNER`, which may only be executed by a superuser.

Notes

The keyword `COLUMN` is noise and can be omitted.

In the current implementation of `ADD COLUMN`, default and NOT NULL clauses for the new column are not supported. The new column always comes into being with all values NULL. You can use the `SET DEFAULT` form of `ALTER TABLE` to set the default afterwards. (You may also want to update the already existing rows to the new default value, using *UPDATE*.) If you want to mark the column non-null, use the `SET NOT NULL` form after you've entered non-null values for the column in all rows.

The `DROP COLUMN` command does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent inserts and updates of the table will store a NULL for the column. Thus, dropping a column is quick but it will not immediately reduce the on-disk size of your table, as the space occupied by the dropped column is not reclaimed. The space will be reclaimed over time as existing rows are updated. To reclaim the space at once, do a dummy `UPDATE` of all rows and then vacuum, as in:

```
UPDATE table SET col = col;
VACUUM FULL table;
```

If a table has any descendant tables, it is not permitted to ADD or RENAME a column in the parent table without doing the same to the descendants --- that is, ALTER TABLE ONLY will be rejected. This ensures that the descendants always have columns matching the parent.

A recursive DROP COLUMN operation will remove a descendant table's column only if the descendant does not inherit that column from any other parents and never had an independent definition of the column. A nonrecursive DROP COLUMN (i.e., ALTER TABLE ONLY ... DROP COLUMN) never removes any descendant columns, but instead marks them as independently defined rather than inherited.

Changing any part of the schema of a system catalog is not permitted.

Refer to `CREATE TABLE` for a further description of valid arguments. The *PostgreSQL User's Guide* has further information on inheritance.

Usage

To add a column of type `varchar` to a table:

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

To drop a column from a table:

```
ALTER TABLE distributors DROP COLUMN address RESTRICT;
```

To rename an existing column:

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

To rename an existing table:

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

To add a NOT NULL constraint to a column:

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

To remove a NOT NULL constraint from a column:

```
ALTER TABLE distributors ALTER COLUMN street DROP NOT NULL;
```

To add a check constraint to a table:

```
    ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);
```

To remove a check constraint from a table and all its children:

```
    ALTER TABLE distributors DROP CONSTRAINT zipchk;
```

To add a foreign key constraint to a table:

```
    ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REF-
ERENCES addresses(address) MATCH FULL;
```

To add a (multicolumn) unique constraint to a table:

```
    ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE (dist_id, zip-
code);
```

To add an automatically named primary key constraint to a table, noting that a table can only ever have one primary key:

```
    ALTER TABLE distributors ADD PRIMARY KEY (dist_id);
```

Compatibility

SQL92

The `ADD COLUMN` form is compliant with the exception that it does not support defaults and NOT NULL constraints, as explained above. The `ALTER COLUMN` form is in full compliance.

The clauses to rename tables, columns, indexes, and sequences are PostgreSQL extensions from SQL92.

ALTER TRIGGER

Name

`ALTER TRIGGER` — change the definition of a trigger

Synopsis

```
ALTER TRIGGER trigger ON table
    RENAME TO newname
```

Inputs

`trigger`

The name of an existing trigger to alter.

`table`

The name of the table on which this trigger acts.

`newname`

New name for the existing trigger.

Outputs

`ALTER TRIGGER`

Message returned from trigger renaming.

`ERROR`

Message returned if trigger is not available, or new name is a duplicate of another existing trigger on the table.

Description

`ALTER TRIGGER` changes the definition of an existing trigger. The `RENAME` clause causes the name of a trigger on the given table to change without otherwise changing the trigger definition.

You must own the table on which the trigger acts in order to change its properties.

Notes

Refer to `CREATE TRIGGER` for a further description of valid arguments.

Usage

To rename an existing trigger:

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

Compatibility

SQL92

The clause to rename triggers is a PostgreSQL extension from SQL92.

ALTER USER

Name

`ALTER USER` — change a database user account

Synopsis

```
ALTER USER username [ [ WITH ] option [ ... ] ]

where option can be:

      [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | CREATEDB | NOCREATEDB
    | CREATEUSER | NOCREATEUSER
    | VALID UNTIL 'abstime'

ALTER USER username SET variable { TO | = } { value | DEFAULT }
ALTER USER username RESET variable
```

Description

`ALTER USER` is used to change the attributes of a PostgreSQL user account. Attributes not mentioned in the command retain their previous settings.

The first variant of this command in the synopsis changes certain global user privileges and authentication settings. (See below for details.) Only a database superuser can change privileges and password expiration with this command. Ordinary users can only change their own password.

The second and the third variant change a user's session default for a specified configuration variable. Whenever the user subsequently starts a new session, the specified value becomes the session default, overriding whatever setting is present in `postgresql.conf` or has been received from the postmaster. Ordinary users can change their own session defaults. Superusers can change anyone's session defaults.

Parameters

`username`

The name of the user whose attributes are to be altered.

`password`

The new password to be used for this account.

`ENCRYPTED`
`UNENCRYPTED`

These key words control whether the password is stored encrypted in `pg_shadow`. (See *CREATE USER* for more information about this choice.)

`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.

`CREATEUSER`
`NOCREATEUSER`

These clauses determine whether a user will be permitted to create new users himself. This option will also make the user a superuser who can override all access restrictions.

`abstime`

The date (and, optionally, the time) at which this user's password is to expire.

`variable`
`value`

Set this user's session default for the specified configuration variable to the given value. If *`value`* is `DEFAULT` or, equivalently, `RESET` is used, the user-specific variable setting is removed and the user will inherit the system-wide default setting in new sessions. Use `RESET ALL` to clear all settings.

See *SET* and the *Administrator's Guide* for more information about allowed variable names and values.

Diagnostics

`ALTER USER`

Message returned if the alteration was successful.

`ERROR: ALTER USER: user "username" does not exist`

Error message returned if the specified user is not known to the database.

Notes

Use *CREATE USER* to add new users, and *DROP USER* to remove a user.

`ALTER USER` cannot change a user's group memberships. Use *ALTER GROUP* to do that.

Using *ALTER DATABASE*, it is also possible to tie a session default to a specific database rather than a user.

Examples

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 authorization 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;
```

Compatibility

The `ALTER USER` statement is a PostgreSQL extension. The SQL standard leaves the definition of users to the implementation.

See Also

CREATE USER, *DROP USER*, *SET*

ANALYZE

Name

`ANALYZE` — collect statistics about a database

Synopsis

```
ANALYZE [ VERBOSE ] [ table [ (column [, ...] ) ] ]
```

Inputs

VERBOSE

Enables display of progress messages.

`table`

The name (possibly schema-qualified) of a specific table to analyze. Defaults to all tables in the current database.

`column`

The name of a specific column to analyze. Defaults to all columns.

Outputs

`ANALYZE`

The command is complete.

Description

`ANALYZE` collects statistics about the contents of PostgreSQL tables, and stores the results in the system table `pg_statistic`. Subsequently, the query planner uses the statistics to help determine the most efficient execution plans for queries.

With no parameter, `ANALYZE` examines every table in the current database. With a parameter, `ANALYZE` examines only that table. It is further possible to give a list of column names, in which case only the statistics for those columns are updated.

Notes

It is a good idea to run `ANALYZE` periodically, or just after making major changes in the contents of a table. Accurate statistics will help the planner to choose the most appropriate query plan, and thereby improve the speed of query processing. A common strategy is to run *VACUUM* and `ANALYZE` once a day during a low-usage time of day.

Unlike `VACUUM FULL`, `ANALYZE` requires only a read lock on the target table, so it can run in parallel with other activity on the table.

For large tables, `ANALYZE` takes a random sample of the table contents, rather than examining every row. This allows even very large tables to be analyzed in a small amount of time. Note however that the statistics are only approximate, and will change slightly each time `ANALYZE` is run, even if the actual table contents did not change. This may result in small changes in the planner's estimated costs shown by `EXPLAIN`.

The collected statistics usually include a list of some of the most common values in each column and a histogram showing the approximate data distribution in each column. One or both of these may be omitted if `ANALYZE` deems them uninteresting (for example, in a unique-key column, there are no common values) or if the column data type does not support the appropriate operators. There is more information about the statistics in the *User's Guide*.

The extent of analysis can be controlled by adjusting the `default_statistics_target` parameter variable, or on a column-by-column basis by setting the per-column statistics target with `ALTER TABLE ALTER COLUMN SET STATISTICS` (see *ALTER TABLE*). The target value sets the maximum number of entries in the most-common-value list and the maximum number of bins in the histogram. The default target value is 10, but this can be adjusted up or down to trade off accuracy of planner estimates against the time taken for `ANALYZE` and the amount of space occupied in `pg_statistic`. In particular, setting the statistics target to zero disables collection of statistics for that column. It may be useful to do that for columns that are never used as part of the WHERE, GROUP BY, or ORDER BY clauses of queries, since the planner will have no use for statistics on such columns.

The largest statistics target among the columns being analyzed determines the number of table rows sampled to prepare the statistics. Increasing the target causes a proportional increase in the time and space needed to do `ANALYZE`.

Compatibility

SQL92

There is no `ANALYZE` statement in SQL92.

BEGIN

Name

`BEGIN` — start a transaction block

Synopsis

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

Inputs

WORK
TRANSACTION

Optional keywords. They have no effect.

Outputs

`BEGIN`

This signifies that a new transaction has been started.

`WARNING: BEGIN: already a transaction in progress`

This indicates that a transaction was already in progress. The current transaction is not affected.

Description

By default, PostgreSQL executes transactions in *unchained mode* (also known as “autocommit” in other database systems). In other words, each user statement is executed in its own transaction and a commit is implicitly performed at the end of the statement (if execution was successful, otherwise a rollback is done). `BEGIN` initiates a user transaction in chained mode, i.e., all user statements after `BEGIN` command will be executed in a single transaction until an explicit *COMMIT* or *ROLLBACK*. Statements are executed more quickly in chained mode, because transaction start/commit requires significant CPU and disk activity. Execution of multiple statements inside a transaction is also useful to ensure consistency when changing several related tables: other clients will be unable to see the intermediate states wherein not all the related updates have been done.

The default transaction isolation level in PostgreSQL is READ COMMITTED, wherein each query inside the transaction sees changes committed before that query begins execution. So, you have to use `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE` just after `BEGIN` if you need more rigorous transaction isolation. (Alternatively, you can change the default transaction isolation level; see the *PostgreSQL Administrator’s Guide* for details.) In SERIALIZABLE mode queries will see only changes committed before the entire transaction began (actually, before execution of the first DML statement in the transaction).

Transactions have the standard ACID (atomic, consistent, isolatable, and durable) properties.

Notes

START TRANSACTION has the same functionality as `BEGIN`.

Use *COMMIT* or *ROLLBACK* to terminate a transaction.

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

If you turn `autocommit` mode off, then `BEGIN` is not required: any SQL command automatically starts a transaction.

Usage

To begin a user transaction:

```
BEGIN WORK;
```

Compatibility

SQL92

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

> **Note:** Many relational database systems offer an autocommit feature as a convenience.

Incidentally, the `BEGIN` keyword is used for a different purpose in embedded SQL. You are advised to be careful about the transaction semantics when porting database applications.

SQL92 also requires SERIALIZABLE to be the default transaction isolation level.

CHECKPOINT

Name

`CHECKPOINT` — force a transaction log checkpoint

Synopsis

```
CHECKPOINT
```

Description

Write-Ahead Logging (WAL) puts a checkpoint in the transaction log every so often. (To adjust the automatic checkpoint interval, see the run-time configuration options *`CHECKPOINT_SEGMENTS`* and *`CHECKPOINT_TIMEOUT`*.) The `CHECKPOINT` command forces an immediate checkpoint when the command is issued, without waiting for a scheduled checkpoint.

A checkpoint is a point in the transaction log sequence at which all data files have been updated to reflect the information in the log. All data files will be flushed to disk. Refer to the *PostgreSQL Administrator's Guide* for more information about the WAL system.

Only superusers may call `CHECKPOINT`. The command is not intended for use during normal operation.

See Also

PostgreSQL Administrator's Guide

Compatibility

The `CHECKPOINT` command is a PostgreSQL language extension.

CLOSE

Name

`CLOSE` — close a cursor

Synopsis

```
CLOSE cursor
```

Inputs

`cursor`

The name of an open cursor to close.

Outputs

`CLOSE CURSOR`

Message returned if the cursor is successfully closed.

`WARNING: 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

PostgreSQL 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` — cluster a table according to an index

Synopsis

```
CLUSTER indexname ON tablename
```

Inputs

`indexname`

The name of an index.

`table`

The name (possibly schema-qualified) of a table.

Outputs

`CLUSTER`

The clustering was done successfully.

Description

`CLUSTER` instructs PostgreSQL to cluster the table specified by *`table`* based on the index specified by *`indexname`*. The index must already have been defined on *`tablename`*.

When a table is clustered, it is physically reordered based on the index information. Clustering is a one-time operation: when the table is subsequently updated, the changes are not clustered. That is, no attempt is made to store new or updated tuples according to their index order. If one wishes, one can periodically re-cluster by issuing the command again.

Notes

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 where `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.

During the cluster operation, a temporary copy of the table is created that contains the table data in the index order. Temporary copies of each index on the table are created as well. Therefore, you need free space on disk at least equal to the sum of the table size and the index sizes.

CLUSTER preserves GRANT, inheritance, index, foreign key, and other ancillary information about the table.

Because the optimizer records statistics about the ordering of tables, it is advisable to run `ANALYZE` on the newly clustered table. Otherwise, the optimizer may make poor choices of query plans.

There is another way to cluster data. The `CLUSTER` command reorders the original table using 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. (PostgreSQL has a cache, but the majority of a big table will not fit in the cache.) The other way to cluster a table is to use

```
SELECT columnlist INTO TABLE newtable
     FROM table ORDER BY columnlist
```

which uses the PostgreSQL sorting code in the ORDER BY clause to create the desired order; this is usually much faster than an index scan for unordered data. You then drop the old table, use `ALTER TABLE...RENAME` to rename *`newtable`* to the old name, and recreate the table's indexes. However, this approach does not preserve OIDs, constraints, foreign key relationships, granted privileges, and other ancillary properties of the table --- all such items must be manually recreated.

Usage

Cluster the employees relation on the basis of its ID attribute:

```
CLUSTER emp_ind ON emp;
```

Compatibility

SQL92

There is no `CLUSTER` statement in SQL92.

COMMENT

Name

`COMMENT` — define or change the comment of an object

Synopsis

```
COMMENT ON
[
  TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE agg_name (agg_type) |
  CONSTRAINT constraint_name ON table_name |
  DATABASE object_name |
  DOMAIN object_name |
  FUNCTION func_name (arg1_type, arg2_type, ...) |
  INDEX object_name |
  OPERATOR op (leftoperand_type, rightoperand_type) |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name
] IS 'text'
```

Inputs

object_name, table_name.column_name, agg_name, constraint_name, func_name, op, rule_name, trigger_name

The name of the object to be be commented. Names of tables, aggregates, domains, functions, indexes, operators, sequences, types, and views may be schema-qualified.

text

The comment to add.

Outputs

`COMMENT`

Message returned if the table is successfully commented.

Description

`COMMENT` stores a comment about a database object. Comments can be easily retrieved with `psql`'s `\dd`, `\d+`, or `\l+` commands. Other user interfaces to retrieve comments can be built atop the same built-in functions that `psql` uses, namely `obj_description()` and `col_description()`.

To modify a comment, issue a new `COMMENT` command for the same object. Only one comment string is stored for each object. To remove a comment, write `NULL` in place of the text string. Comments are automatically dropped when the object is dropped.

> **Note:** There is presently no security mechanism for comments: any user connected to a database can see all the comments for objects in that database (although only superusers can change comments for objects that they don't own). Therefore, don't put security-critical information in comments.

Usage

Attach a comment to the table `mytable`:

```
COMMENT ON TABLE mytable IS 'This is my table.';
```

Remove it again:

```
COMMENT ON TABLE mytable IS NULL;
```

Some more examples:

```
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes sample vari-
ance';
COMMENT ON COLUMN my_table.my_field IS 'Employee ID number';
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON DOMAIN my_domain IS 'Email Address Domain';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman Numeral';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee id';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two texts';
COMMENT ON OPERATOR ^ (NONE, text) IS 'This is a prefix operator on text';
COMMENT ON RULE my_rule ON my_table IS 'Logs UPDATES of employee records';
COMMENT ON SCHEMA my_schema IS 'Departmental data';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON TABLE my_schema.my_table IS 'Employee Information';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for R.I.';
COMMENT ON TYPE complex IS 'Complex Number datatype';
COMMENT ON VIEW my_view IS 'View of departmental costs';
```

Compatibility

SQL92

There is no `COMMENT` in SQL92.

COMMIT

Name

`COMMIT` — commit the current transaction

Synopsis

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

Inputs

WORK
TRANSACTION

Optional keywords. They have no effect.

Outputs

`COMMIT`

Message returned if the transaction is successfully committed.

`WARNING: COMMIT: no transaction in progress`

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 *ROLLBACK* to abort a transaction.

Usage

To make all changes permanent:

```
COMMIT WORK;
```

Compatibility

SQL92

SQL92 only specifies the two forms `COMMIT` and `COMMIT WORK`. Otherwise full compatibility.

COPY

Name

`COPY` — copy data between files and tables

Synopsis

```
COPY table [ ( column [, ...] ) ]
    FROM { 'filename' | stdin }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter' ]
          [ NULL [ AS ] 'null string' ] ]
COPY table [ ( column [, ...] ) ]
    TO { 'filename' | stdout }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter' ]
          [ NULL [ AS ] 'null string' ] ]
```

Inputs

`table`

The name (possibly schema-qualified) of an existing table.

`column`

An optional list of columns to be copied. If no column list is specified, all columns will be used.

`filename`

The absolute Unix path name of the input or output file.

`stdin`

Specifies that input comes from the client application.

`stdout`

Specifies that output goes to the client application.

BINARY

Changes the behavior of field formatting, forcing all data to be stored or read in binary format rather than as text. You can not specify `DELIMITER` or `NULL` in binary mode.

OIDS

Specifies copying the internal object id (OID) for each row.

`delimiter`

The single character that separates fields within each row (line) of the file.

`null string`

The string that represents a NULL value. The default is “`\N`” (backslash-N). You might prefer an empty string, for example.

> **Note:** On a copy in, any data item that matches this string will be stored as a NULL value, so you should make sure that you use the same string as you used on copy out.

Outputs

`COPY`

The copy completed successfully.

`ERROR:` *`reason`*

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

Description

`COPY` moves data between PostgreSQL tables and standard file-system files. `COPY TO` copies the contents of a table *to* a file, while `COPY FROM` copies data *from* a file to a table (appending the data to whatever is in the table already).

If a list of columns is specified, `COPY` will only copy the data in the specified columns to or from the file. If there are any columns in the table that are not in the column list, `COPY FROM` will insert the default values for those columns.

`COPY` with a file name instructs the PostgreSQL backend to directly read from or write to a file. The file must be accessible to the backend and the name must be specified from the viewpoint of the backend. When `stdin` or `stdout` is specified, data flows through the client frontend to the backend.

> **Tip:** Do not confuse `COPY` with the psql instruction `\copy`. `\copy` invokes `COPY FROM stdin` or `COPY TO stdout`, and then fetches/stores the data in a file accessible to the psql client. Thus, file accessibility and access rights depend on the client rather than the backend when `\copy` is used.

Notes

`COPY` can only be used with plain tables, not with views.

The BINARY keyword will force all data to be stored/read as binary format rather than as text. It is somewhat faster than the normal copy command, but a binary copy file is not portable across machine architectures.

By default, a text copy uses a tab ("\t") character as a delimiter between fields. The field delimiter may be changed to any other single character with the keyword `DELIMITER`. Characters in data fields that happen to match the delimiter character will be backslash quoted.

You must have *select privilege* on any table whose values are read by `COPY TO`, and *insert privilege* on a table into which values are being inserted by `COPY FROM`. The backend also needs appropriate Unix permissions for any file read or written by `COPY`.

`COPY FROM` will invoke any triggers and check constraints on the destination table. However, it will not invoke rules.

`COPY` stops operation at the first error. This should not lead to problems in the event of a `COPY TO`, but the target relation will already have received earlier rows in a `COPY FROM`. These rows will not be visible or accessible, but they still occupy disk space. This may amount to a considerable amount of wasted disk space if the failure happened well into a large copy operation. You may wish to invoke `VACUUM` to recover the wasted space.

Files named in a `COPY` command are read or written directly by the backend, not by the client application. Therefore, they must reside on or be accessible to the database server machine, not the client. They must be accessible to and readable or writable by the PostgreSQL user (the user ID the server runs as), not the client. `COPY` naming a file is only allowed to database superusers, since it allows reading or writing any file that the backend has privileges to access.

Tip: The psql instruction `\copy` reads or writes files on the client machine with the client's permissions, so it is not restricted to superusers.

It is recommended that the file name used in `COPY` always be specified as an absolute path. This is enforced by the backend in the case of `COPY TO`, but for `COPY FROM` you do have the option of reading from a file specified by a relative path. The path will be interpreted relative to the backend's working directory (somewhere below `$PGDATA`), not the client's working directory.

File Formats

Text Format

When `COPY` is used without the BINARY option, the file read or written is a text file with one line per table row. Columns (attributes) in a row are separated by the delimiter character. The attribute values themselves are strings generated by the output function, or acceptable to the input function, of each attribute's data type. The specified null-value string is used in place of attributes that are NULL. `COPY FROM` will raise an error if any line of the input file contains more or fewer columns than are expected.

If OIDS is specified, the OID is read or written as the first column, preceding the user data columns. (An error is raised if OIDS is specified for a table that does not have OIDs.)

End of data can be represented by a single line containing just backslash-period (`\.`). An end-of-data marker is not necessary when reading from a Unix file, since the end of file serves perfectly well; but an end marker must be provided when copying data to or from a client application.

Backslash characters (\) may be used in the `COPY` data to quote data characters that might otherwise be taken as row or column delimiters. In particular, the following characters *must* be preceded by a backslash if they appear as part of an attribute value: backslash itself, newline, and the current delimiter character.

The following special backslash sequences are recognized by `COPY FROM`:

Sequence	**Represents**
`\b`	Backspace (ASCII 8)
`\f`	Form feed (ASCII 12)
`\n`	Newline (ASCII 10)
`\r`	Carriage return (ASCII 13)
`\t`	Tab (ASCII 9)
`\v`	Vertical tab (ASCII 11)
`\`*`digits`*	Backslash followed by one to three octal digits specifies the character with that numeric code

Presently, `COPY TO` will never emit an octal-digits backslash sequence, but it does use the other sequences listed above for those control characters.

Never put a backslash before a data character `N` or period (`.`). Such pairs will be mistaken for the default null string or the end-of-data marker, respectively. Any other backslashed character that is not mentioned in the above table will be taken to represent itself.

It is strongly recommended that applications generating COPY data convert data newlines and carriage returns to the `\n` and `\r` sequences respectively. At present (PostgreSQL 7.2 and older versions) it is possible to represent a data carriage return without any special quoting, and to represent a data newline by a backslash and newline. However, these representations will not be accepted by default in future releases.

Note that the end of each row is marked by a Unix-style newline ("\n"). Presently, `COPY FROM` will not behave as desired if given a file containing DOS- or Mac-style newlines. This is expected to change in future releases.

Binary Format

The file format used for `COPY BINARY` changed in PostgreSQL v7.1. The new format consists of a file header, zero or more tuples, and a file trailer.

File Header

The file header consists of 24 bytes of fixed fields, followed by a variable-length header extension area. The fixed fields are:

Signature

12-byte sequence `PGBCOPY\n\377\r\n\0` --- note that the null is a required part of the signature. (The signature is designed to allow easy identification of files that have been munged by a non-8-bit-clean transfer. This signature will be changed by newline-translation filters, dropped nulls, dropped high bits, or parity changes.)

Integer layout field

int32 constant 0x01020304 in source's byte order. Potentially, a reader could engage in byte-flipping of subsequent fields if the wrong byte order is detected here.

Flags field

int32 bit mask to denote important aspects of the file format. Bits are numbered from 0 (LSB) to 31 (MSB) --- note that this field is stored with source's endianness, as are all subsequent integer fields. Bits 16-31 are reserved to denote critical file format issues; a reader should abort if it finds an unexpected bit set in this range. Bits 0-15 are reserved to signal backwards-compatible format issues; a reader should simply ignore any unexpected bits set in this range. Currently only one flag bit is defined, and the rest must be zero:

Bit 16

if 1, OIDs are included in the dump; if 0, not

Header extension area length

int32 length in bytes of remainder of header, not including self. In the initial version this will be zero, and the first tuple follows immediately. Future changes to the format might allow additional data to be present in the header. A reader should silently skip over any header extension data it does not know what to do with.

The header extension area is envisioned to contain a sequence of self-identifying chunks. The flags field is not intended to tell readers what is in the extension area. Specific design of header extension contents is left for a later release.

This design allows for both backwards-compatible header additions (add header extension chunks, or set low-order flag bits) and non-backwards-compatible changes (set high-order flag bits to signal such changes, and add supporting data to the extension area if needed).

Tuples

Each tuple begins with an int16 count of the number of fields in the tuple. (Presently, all tuples in a table will have the same count, but that might not always be true.) Then, repeated for each field in the tuple, there is an int16 `typlen` word possibly followed by field data. The `typlen` field is interpreted thus:

Zero

Field is NULL. No data follows.

>0

Field is a fixed-length data type. Exactly N bytes of data follow the `typlen` word.

-1

Field is a `varlena` data type. The next four bytes are the `varlena` header, which contains the total value length including itself.

$<$ -1

Reserved for future use.

For non-NULL fields, the reader can check that the `typlen` matches the expected `typlen` for the destination column. This provides a simple but very useful check that the data is as expected.

There is no alignment padding or any other extra data between fields. Note also that the format does not distinguish whether a data type is pass-by-reference or pass-by-value. Both of these provisions are deliberate: they might help improve portability of the files (although of course endianness and floating-point-format issues can still keep you from moving a binary file across machines).

If OIDs are included in the dump, the OID field immediately follows the field-count word. It is a normal field except that it's not included in the field-count. In particular it has a `typlen` --- this will allow handling of 4-byte vs 8-byte OIDs without too much pain, and will allow OIDs to be shown as NULL if that ever proves desirable.

File Trailer

The file trailer consists of an int16 word containing -1. This is easily distinguished from a tuple's field-count word.

A reader should report an error if a field-count word is neither -1 nor the expected number of columns. This provides an extra check against somehow getting out of sync with the data.

Usage

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

```
COPY country TO stdout WITH DELIMITER '|';
```

To copy data from a Unix file into the `country` table:

```
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
\.
```

Note that the white space on each line is actually a TAB.

The following is 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)`, the second is `text`, and the third is `integer`. All the rows have a null value in the third field.

```
0000000   P   G   B   C   O   P   Y  \n 377  \r  \n  \0 004 003 002 001
0000020  \0  \0  \0  \0  \0  \0  \0  \0 003  \0 377 377 006  \0  \0  \0
0000040   A   F 377 377 017  \0  \0  \0   A   F   G   H   A   N   I   S
0000060   T   A   N  \0  \0 003  \0 377 377 006  \0  \0  \0   A   L 377
0000100 377  \v  \0  \0  \0   A   L   B   A   N   I   A  \0  \0 003  \0
0000120 377 377 006  \0  \0  \0   D   Z 377 377  \v  \0  \0  \0   A   L
0000140   G   E   R   I   A  \0  \0 003  \0 377 377 006  \0  \0  \0   Z
```

```
0000160   M 377 377  \n  \0  \0  \0   Z   A   M   B   I   A  \0  \0 003
0000200  \0 377 377 006  \0  \0  \0   Z   W 377 377  \f  \0  \0  \0   Z
0000220   I   M   B   A   B   W   E  \0  \0 377 377
```

Compatibility

SQL92

There is no `COPY` statement in SQL92.

The following syntax was used by pre-7.3 applications and is still supported:

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

CREATE AGGREGATE

Name

`CREATE AGGREGATE` — define a new aggregate function

Synopsis

```
CREATE AGGREGATE name ( BASETYPE = input_data_type,
    SFUNC = sfunc, STYPE = state_type
    [ , FINALFUNC = ffunc ]
    [ , INITCOND = initial_condition ] )
```

Inputs

`name`

The name (optionally schema-qualified) of an aggregate function to create.

`input_data_type`

The input data type on which this aggregate function operates. This can be specified as `"ANY"` for an aggregate that does not examine its input values (an example is `count(*)`).

`sfunc`

The name of the state transition function to be called for each input data value. This is normally a function of two arguments, the first being of type *`state_type`* and the second of type *`input_data_type`*. Alternatively, for an aggregate that does not examine its input values, the function takes just one argument of type *`state_type`*. In either case the function must return a value of type *`state_type`*. This function takes the current state value and the current input data item, and returns the next state value.

`state_type`

The data type for the aggregate's state value.

`ffunc`

The name of the final function called to compute the aggregate's result after all input data has been traversed. The function must take a single argument of type *`state_type`*. The output data type of the aggregate is defined as the return type of this function. If *`ffunc`* is not specified, then the ending state value is used as the aggregate's result, and the output type is *`state_type`*.

`initial_condition`

The initial setting for the state value. This must be a literal constant in the form accepted for the data type *`state_type`*. If not specified, the state value starts out NULL.

Outputs

`CREATE AGGREGATE`

Message returned if the command completes successfully.

Description

`CREATE AGGREGATE` allows a user or programmer to extend PostgreSQL functionality by defining new aggregate functions. Some aggregate functions for base types such as `min(integer)` and `avg(double precision)` 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.

If a schema name is given (for example, `CREATE AGGREGATE myschema.myagg ...`) then the aggregate function is created in the specified schema. Otherwise it is created in the current schema (the one at the front of the search path; see `CURRENT_SCHEMA()`).

An aggregate function is identified by its name and input data type. Two aggregates in the same schema can have the same name if they operate on different input types. The name and input data type of an aggregate must also be distinct from the name and input data type(s) of every ordinary function in the same schema.

An aggregate function is made from one or two ordinary functions: a state transition function *`sfunc`*, and an optional final calculation function *`ffunc`*. These are used as follows:

```
sfunc( internal-state, next-data-item ) ---> next-internal-state
ffunc( internal-state ) ---> aggregate-value
```

PostgreSQL creates a temporary variable of data type *`stype`* to hold the current internal state of the aggregate. At each input data item, the state transition function is invoked to calculate a new internal state value. After all the data has been processed, the final function is invoked once to calculate the aggregate's output value. If there is no final function then the ending state value is returned as-is.

An aggregate function may provide an initial condition, that is, an initial value for the internal state value. This is specified and stored in the database as a field of type `text`, but it must be a valid external representation of a constant of the state value data type. If it is not supplied then the state value starts out NULL.

If the state transition function is declared "strict", then it cannot be called with NULL inputs. With such a transition function, aggregate execution behaves as follows. NULL input values are ignored (the function is not called and the previous state value is retained). If the initial state value is NULL, then the first non-NULL input value replaces the state value, and the transition function is invoked beginning with the second non-NULL input value. This is handy for implementing aggregates like `max`. Note that this behavior is only available when *`state_type`* is the same as *`input_data_type`*. When these types are different, you must supply a non-NULL initial condition or use a non-strict transition function.

If the state transition function is not strict, then it will be called unconditionally at each input value, and must deal with NULL inputs and NULL transition values for itself. This allows the aggregate author to have full control over the aggregate's handling of null values.

If the final function is declared "strict", then it will not be called when the ending state value is NULL; instead a NULL result will be output automatically. (Of course this is just the normal behavior of strict functions.) In any case the final function has the option of returning NULL. For example, the final function for `avg` returns NULL when it sees there were zero input tuples.

Notes

Use `DROP AGGREGATE` to drop aggregate functions.

The parameters of `CREATE AGGREGATE` can be written in any order, not just the order illustrated above.

Usage

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

Compatibility

SQL92

`CREATE AGGREGATE` is a PostgreSQL language extension. There is no `CREATE AGGREGATE` in SQL92.

CREATE CAST

Name

`CREATE CAST` — define a user-defined cast

Synopsis

```
CREATE CAST (sourcetype AS targettype)
    WITH FUNCTION funcname (argtype)
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (sourcetype AS targettype)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

Description

`CREATE CAST` defines a new cast. A cast specifies how to perform a conversion between two data types. For example,

```
SELECT CAST(42 AS text);
```

converts the integer constant 42 to type `text` by invoking a previously specified function, in this case `text(int4)`. (If no suitable cast has been defined, the conversion fails.)

Two types may be *binary compatible*, which means that they can be converted into one another "for free" without invoking any function. This requires that corresponding values use the same internal representation. For instance, the types `text` and `varchar` are binary compatible.

By default, a cast can be invoked only by an explicit cast request, that is an explicit `CAST(`*`x`*` AS `*`typename`*`)`, *`x`*`::`*`typename`*, or *`typename`*`(`*`x`*`)` construct.

If the cast is marked `AS ASSIGNMENT` then it can be invoked implicitly when assigning to a column of the target data type. For example, supposing that `foo.f1` is a column of type `text`, then

```
INSERT INTO foo(f1) VALUES(42);
```

will be allowed if the cast from type `integer` to type `text` is marked `AS ASSIGNMENT`, otherwise not. (We generally use the term *assignment cast* to describe this kind of cast.)

If the cast is marked `AS IMPLICIT` then it can be invoked implicitly in any context, whether assignment or internally in an expression. For example, since `||` takes `text` arguments,

```
SELECT 'The time is ' || now();
```

will be allowed only if the cast from type `timestamp` to `text` is marked `AS IMPLICIT`. Otherwise it will be necessary to write the cast explicitly, for example

```
SELECT 'The time is ' || CAST(now() AS text);
```

(We generally use the term *implicit cast* to describe this kind of cast.)

It is wise to be conservative about marking casts as implicit. An overabundance of implicit casting paths can cause PostgreSQL to choose surprising interpretations of commands, or to be unable to resolve commands at all because there are multiple possible interpretations. A good rule of thumb is to make a cast implicitly invokable only for information-preserving transformations between types in the same general type category. For example, the cast from `int2` to `int4` can reasonably be implicit, but the cast from `float8` to `int4` should probably be assignment-only. Cross-type-category casts, such as `text` to `int4`, are best made explicit-only.

To be able to create a cast, you must own the source or the target data type. To create a binary-compatible cast, you must be superuser (this restriction is made because an erroneous binary-compatible cast conversion can easily crash the server).

Parameters

`sourcetype`

The name of the source data type of the cast.

`targettype`

The name of the target data type of the cast.

`funcname`(*`argtype`*)

The function used to perform the cast. The function name may be schema-qualified. If it is not, the function will be looked up in the path. The argument type must be identical to the source type, the result data type must match the target type of the cast.

`WITHOUT FUNCTION`

Indicates that the source type and the target type are binary compatible, so no function is required to perform the cast.

`AS ASSIGNMENT`

Indicates that the cast may be invoked implicitly in assignment contexts.

`AS IMPLICIT`

Indicates that the cast may be invoked implicitly in any context.

Notes

Use `DROP CAST` to remove user-defined casts.

Remember that if you want to be able to convert types both ways you need to declare casts both ways explicitly.

Prior to PostgreSQL 7.3, every function that had the same name as a data type, returned that data type, and took one argument of a different type was automatically a cast function. This convention has been abandoned in face of the introduction of schemas and to be able to represent binary compatible casts in the catalogs. (The built-in cast functions still follow this naming scheme, but they have to be shown as casts in `pg_cast` now.)

Examples

To create a cast from type `text` to type `int4` using the function `int4(text)`:

```
CREATE CAST (text AS int4) WITH FUNCTION int4(text);
```

(This cast is already predefined in the system.)

Compatibility

The `CREATE CAST` command conforms to SQL99, except that SQL99 does not make provisions for binary compatible types. `AS IMPLICIT` is a PostgreSQL extension, too.

See Also

CREATE FUNCTION, *CREATE TYPE*, *DROP CAST*, *PostgreSQL Programmer's Guide*

CREATE CONSTRAINT TRIGGER

Name

`CREATE CONSTRAINT TRIGGER` — define a new constraint trigger

Synopsis

```
CREATE CONSTRAINT TRIGGER name
    AFTER events ON
    relation constraint attributes
    FOR EACH ROW EXECUTE PROCEDURE func '(' args ')'
```

Inputs

name

The name of the constraint trigger.

events

The event categories for which this trigger should be fired.

relation

The name (possibly schema-qualified) of the relation in which the triggering events occur.

constraint

Actual constraint specification.

attributes

Constraint attributes.

func(*args*)

Function to call as part of the trigger processing.

Outputs

`CREATE TRIGGER`

Message returned if successful.

Description

`CREATE CONSTRAINT TRIGGER` is used within `CREATE/ALTER TABLE` and by pg_dump to create the special triggers for referential integrity.

It is not intended for general use.

CREATE CONVERSION

Name

`CREATE CONVERSION` — define a user-defined conversion

Synopsis

```
CREATE [DEFAULT] CONVERSION conversion_name
    FOR source_encoding TO dest_encoding FROM funcname
```

Description

`CREATE CONVERSION` defines a new encoding conversion. Conversion names may be used in the CONVERT() function to specify a particular encoding conversion. Also, conversions that are marked DEFAULT can be used for automatic encoding conversion between frontend and backend. For this purpose, two conversions, from encoding A to B AND from encoding B to A, must be defined.

To be able to create a conversion, you must have the execute right on the function and the create right on the destination schema.

Parameters

`DEFAULT`

The `DEFAULT` clause indicates that this conversion is the default for this particular source to destination encoding. There should be only one default encoding in a schema for the encoding pair.

`conversion_name`

The name of the conversion. The conversion name may be schema-qualified. If it is not, the conversion is defined in the current schema. The conversion name must be unique within a schema.

`source_encoding`

The source encoding name.

`source_encoding`

The destination encoding name.

`funcname`

The function used to perform the conversion. The function name may be schema-qualified. If it is not, the function will be looked up in the path.

The function must have the following signature:

```
conv_proc(
INTEGER, -- source encoding id
INTEGER, -- destination encoding id
CSTRING, -- source string (null terminated C string)
CSTRING, -- destination string (null terminated C string)
INTEGER  -- source string length
) returns VOID;
```

Notes

Use `DROP CONVERSION` to remove user-defined conversions.

The privileges required to create a conversion may be changed in a future release.

Examples

To create a conversion from encoding UNICODE to LATIN1 using `myfunc`:

```
CREATE CONVERSION myconv FOR 'UNICODE' TO 'LATIN1' FROM myfunc;
```

Compatibility

`CREATE CONVERSION` is a PostgreSQL extension. There is no `CREATE CONVERSION` statement in SQL99.

See Also

CREATE FUNCTION, *DROP CONVERSION*, *PostgreSQL Programmer's Guide*

CREATE DATABASE

Name

`CREATE DATABASE` — create a new database

Synopsis

```
CREATE DATABASE name
    [ [ WITH ] [ OWNER [=] dbowner ]
           [ LOCATION [=] 'dbpath' ]
           [ TEMPLATE [=] template ]
           [ ENCODING [=] encoding ] ]
```

Inputs

name

The name of a database to create.

dbowner

Name of the database user who will own the new database, or `DEFAULT` to use the default (namely, the user executing the command).

dbpath

An alternate file-system location in which to store the new database, specified as a string literal; or `DEFAULT` to use the default location.

template

Name of template from which to create the new database, or `DEFAULT` to use the default template (`template1`).

encoding

Multibyte encoding method to use in the new database. Specify a string literal name (e.g., `'SQL_ASCII'`), or an integer encoding number, or `DEFAULT` to use the default encoding.

Outputs

`CREATE DATABASE`

Message returned if the command completes successfully.

`ERROR: user 'username' is not allowed to create/drop databases`

You must have the special `CREATEDB` privilege to create databases. See *CREATE USER*.

`ERROR: createdb: database "name" already exists`

This occurs if a database with the *name* specified already exists.

`ERROR: database path may not contain single quotes`

The database location *`dbpath`* cannot contain single quotes. This is required so that the shell commands that create the database directory can execute safely.

`ERROR: CREATE DATABASE: may not be called in a transaction block`

If you have an explicit transaction block in progress you cannot call `CREATE DATABASE`. You must finish the transaction first.

`ERROR: Unable to create database directory 'path'.`
`ERROR: Could not initialize database directory.`

These are most likely related to insufficient permissions on the data directory, a full disk, or other file system problems. The user under which the database server is running must have access to the location.

Description

`CREATE DATABASE` creates a new PostgreSQL database.

Normally, the creator becomes the owner of the new database. Superusers can create databases owned by other users using the `OWNER` clause. They can even create databases owned by users with no special privileges. Non-superusers with `CREATEDB` privilege can only create databases owned by themselves.

An alternate location can be specified in order to, for example, store the database on a different disk. The path must have been prepared with the *initlocation* command.

If the path name does not contain a slash, it is interpreted as an environment variable name, which must be known to the server process. This way the database administrator can exercise control over locations in which databases can be created. (A customary choice is, e.g., `PGDATA2`.) If the server is compiled with `ALLOW_ABSOLUTE_DBPATHS` (not so by default), absolute path names, as identified by a leading slash (e.g., `/usr/local/pgsql/data`), are allowed as well.

By default, the new database will be created by cloning the standard system database `template1`. A different template can be specified by writing `TEMPLATE =` *`name`*. In particular, by writing `TEMPLATE = template0`, you can create a virgin database containing only the standard objects predefined by your version of PostgreSQL. This is useful if you wish to avoid copying any installation-local objects that may have been added to `template1`.

The optional encoding parameter allows selection of the database encoding, if your server was compiled with multibyte encoding support. When not specified, it defaults to the encoding used by the selected template database.

Optional parameters can be written in any order, not only the order illustrated above.

Notes

`CREATE DATABASE` is a PostgreSQL language extension.

Use *DROP DATABASE* to remove a database.

The program *createdb* is a shell script wrapper around this command, provided for convenience.

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.

Although it is possible to copy a database other than `template1` by specifying its name as the template, this is not (yet) intended as a general-purpose COPY DATABASE facility. We recommend that databases used as templates be treated as read-only. See the *Administrator's Guide* for more information.

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
        The location will be initialized with username "olly".
This user will own all the files and must also own the server process.
Creating directory /home/olly/private_db
Creating directory /home/olly/private_db/base

initlocation is complete.


    $ psql olly
    Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

olly=> CREATE DATABASE elsewhere WITH LOCATION = '/home/olly/private_db';
    CREATE DATABASE
```

Compatibility

SQL92

There is no `CREATE DATABASE` statement in SQL92. Databases are equivalent to catalogs, whose creation is implementation-defined.

CREATE DOMAIN

Name

`CREATE DOMAIN` — define a new domain

Synopsis

```
CREATE DOMAIN domainname [AS] data_type
    [ DEFAULT default_expr ]
    [ constraint [, ... ] ]

where constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL }
```

Parameters

`domainname`

The name (optionally schema-qualified) of a domain to be created.

`data_type`

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

`DEFAULT` *`default_expr`*

The `DEFAULT` clause specifies a default value for columns of the domain data type. The value is any variable-free expression (but subselects are not allowed). The data type of the default expression must match the data type of the domain.

The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a domain, then the default is NULL.

> **Note:** If a default value is specified for a particular column, it overrides any default associated with the domain. In turn, the domain default overrides any default value associated with the underlying data type.

`CONSTRAINT` *`constraint_name`*

An optional name for a constraint. If not specified, the system generates a name.

`NOT NULL`

Values of this domain are not allowed to be NULL.

`NULL`

Values of this domain are allowed to be NULL. This is the default.

This clause is only available for compatibility with non-standard SQL databases. Its use is discouraged in new applications.

Outputs

`CREATE DOMAIN`

Message returned if the domain is successfully created.

Description

`CREATE DOMAIN` allows the user to register a new data domain with PostgreSQL for use in the current data base. The user who defines a domain becomes its owner.

If a schema name is given (for example, `CREATE DOMAIN myschema.mydomain ...`) then the domain is created in the specified schema. Otherwise it is created in the current schema (the one at the front of the search path; see `CURRENT_SCHEMA()`). The domain name must be unique among the types and domains existing in its schema.

Domains are useful for abstracting common fields between tables into a single location for maintenance. An email address column may be used in several tables, all with the same properties. Define a domain and use that rather than setting up each table's constraints individually.

Examples

This example creates the `country_code` data type and then uses the type in a table definition:

```
CREATE DOMAIN country_code char(2) NOT NULL;
CREATE TABLE countrylist (id INT4, country country_code);
```

Compatibility

SQL99 defines CREATE DOMAIN, but says that the only allowed constraint type is CHECK constraints. CHECK constraints for domains are not yet supported by PostgreSQL.

See Also

DROP DOMAIN, *PostgreSQL Programmer's Guide*

CREATE FUNCTION

Name

`CREATE FUNCTION` — define a new function

Synopsis

```
CREATE [ OR REPLACE ] FUNCTION name ( [ argtype [, ...] ] )
    RETURNS rettype
  { LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
    [ WITH ( attribute [, ...] ) ]
```

Description

`CREATE FUNCTION` defines a new function. `CREATE OR REPLACE FUNCTION` will either create a new function, or replace an existing definition.

The user that creates the function becomes the owner of the function.

Parameters

`name`

The name of a function to create. If a schema name is included, then the function is created in the specified schema. Otherwise it is created in the current schema (the one at the front of the search path; see `CURRENT_SCHEMA()`). The name of the new function must not match any existing function with the same argument types in the same schema. However, functions of different argument types may share a name (this is called *overloading*).

`argtype`

The data type(s) of the function's arguments, if any. The input types may be base, complex, or domain types, or the same as the type of an existing column. The type of a column is referenced by writing *`tablename.columnname`*`%TYPE`; using this can sometimes help make a function independent from changes to the definition of a table. Depending on the implementation language it may also be allowed to specify "pseudo-types" such as `cstring`. Pseudo-types indicate that the actual argument type is either incompletely specified, or outside the set of ordinary SQL data types.

`rettype`

The return data type. The return type may be specified as a base, complex, or domain type, or the same as the type of an existing column. Depending on the implementation language it may also be allowed to specify "pseudo-types" such as `cstring`. The `setof` modifier indicates that the function will return a set of items, rather than a single item.

`langname`

The name of the language that the function is implemented in. May be `SQL`, `C`, `internal`, or the name of a user-defined procedural language. (See also *createlang*.) For backward compatibility, the name may be enclosed by single quotes.

IMMUTABLE
STABLE
VOLATILE

These attributes inform the system whether it is safe to replace multiple evaluations of the function with a single evaluation, for run-time optimization. At most one choice should be specified. If none of these appear, `VOLATILE` is the default assumption.

`IMMUTABLE` indicates that the function always returns the same result when given the same argument values; that is, it does not do database lookups or otherwise use information not directly present in its parameter list. If this option is given, any call of the function with all-constant arguments can be immediately replaced with the function value.

`STABLE` indicates that within a single table scan the function will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for functions whose results depend on database lookups, parameter variables (such as the current time zone), etc. Also note that the `CURRENT_TIMESTAMP` family of functions qualify as stable, since their values do not change within a transaction.

`VOLATILE` indicates that the function value can change even within a single table scan, so no optimizations can be made. Relatively few database functions are volatile in this sense; some examples are `random()`, `currval()`, `timeofday()`. Note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away; an example is `setval()`.

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

`CALLED ON NULL INPUT` (the default) indicates that the function will be called normally when some of its arguments are null. It is then the function author's responsibility to check for null values if necessary and respond appropriately.

`RETURNS NULL ON NULL INPUT` or `STRICT` indicates that the function always returns NULL whenever any of its arguments are NULL. If this parameter is specified, the function is not executed when there are NULL arguments; instead a NULL result is assumed automatically.

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

`SECURITY INVOKER` indicates that the function is to be executed with the privileges of the user that calls it. That is the default. `SECURITY DEFINER` specifies that the function is to be executed with the privileges of the user that created it.

The key word `EXTERNAL` is present for SQL compatibility but is optional since, unlike in SQL, this feature does not only apply to external functions.

`definition`

A string defining the function; the meaning depends on the language. It may be an internal function name, the path to an object file, an SQL query, or text in a procedural language.

`obj_file`, `link_symbol`

This form of the `AS` clause is used for dynamically linked C language functions when the function name in the C language source code is not the same as the name of the SQL function. The string `obj_file` is the name of the file containing the dynamically loadable object, and `link_symbol` is the object's link symbol, that is, the name of the function in the C language source code.

`attribute`

The historical way to specify optional pieces of information about the function. The following attributes may appear here:

`isStrict`

Equivalent to `STRICT` or `RETURNS NULL ON NULL INPUT`

`isCachable`

`isCachable` is an obsolete equivalent of `IMMUTABLE`; it's still accepted for backwards-compatibility reasons.

Attribute names are not case-sensitive.

Notes

Refer to the chapter in the *PostgreSQL Programmer's Guide* on the topic of extending PostgreSQL via functions for further information on writing external functions.

The full SQL type syntax is allowed for input arguments and return value. However, some details of the type specification (e.g., the precision field for `numeric` types) are the responsibility of the underlying function implementation and are silently swallowed (i.e., not recognized or enforced) by the `CREATE FUNCTION` command.

PostgreSQL allows function *overloading*; that is, the same name can be used for several different functions so long as they have distinct argument types. This facility must be used with caution for internal and C-language functions, however.

Two `internal` functions cannot have the same C name without causing errors at link time. To get around that, give them different C names (for example, use the argument types as part of the C names), then specify those names in the AS clause of `CREATE FUNCTION`. If the AS clause is left empty, then `CREATE FUNCTION` assumes the C name of the function is the same as the SQL name.

Similarly, when overloading SQL function names with multiple C-language functions, give each C-language instance of the function a distinct name, then use the alternative form of the `AS` clause in the `CREATE FUNCTION` syntax to select the appropriate C-language implementation of each overloaded SQL function.

When repeated `CREATE FUNCTION` calls refer to the same object file, the file is only loaded once. To unload and reload the file (perhaps during development), use the *LOAD* command.

Use `DROP FUNCTION` to remove user-defined functions.

To update the definition of an existing function, use `CREATE OR REPLACE FUNCTION`. Note that it is not possible to change the name or argument types of a function this way (if you tried, you'd just be creating a new, distinct function). Also, `CREATE OR REPLACE FUNCTION` will not let you change the return type of an existing function. To do that, you must drop and re-create the function.

If you drop and then re-create a function, the new function is not the same entity as the old; you will break existing rules, views, triggers, etc that referred to the old function. Use `CREATE OR REPLACE FUNCTION` to change a function definition without breaking objects that refer to the function.

To be able to define a function, the user must have the `USAGE` privilege on the language.

By default, only the owner (creator) of the function has the right to execute it. Other users must be granted the `EXECUTE` privilege on the function to be able to use it.

Examples

To create a simple SQL function:

```
    CREATE FUNCTION one() RETURNS integer
        AS ’SELECT 1 AS RESULT;’
        LANGUAGE SQL;

    SELECT one() AS answer;
     answer
--------
      1
```

The next example creates a C function by calling a routine from a user-created shared library named `funcs.so` (the extension may vary across platforms). The shared library file is sought in the server’s dynamic library search path. 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 constraint.

```
    CREATE FUNCTION ean_checkdigit(char, char) RETURNS boolean
        AS ’funcs’ 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))
    );
```

The next example creates a function that does type conversion from the user-defined type complex to the built-in type point. The function is implemented by a dynamically loaded object that was compiled from C source (we illustrate the now-deprecated alternative of specifying the absolute file name to the shared object file). For PostgreSQL to find a type conversion function automatically, the SQL function has to have the same name as the return type, and so overloading is unavoidable. The function name is overloaded by using the second form of the `AS` clause in the SQL definition:

```
    CREATE FUNCTION point(complex) RETURNS point
        AS ’/home/bernie/pgsql/lib/complex.so’, ’complex_to_point’
        LANGUAGE C STRICT;
```

The C declaration of the function could be:

```
    Point * complex_to_point (Complex *z)
    {
     Point *p;

     p = (Point *) palloc(sizeof(Point));
     p->x = z->x;
     p->y = z->y;

     return p;
    }
```

Note that the function is marked “strict”; this allows us to skip checking for NULL input in the function body.

Compatibility

A `CREATE FUNCTION` command is defined in SQL99. The PostgreSQL version is similar but not fully compatible. The attributes are not portable, neither are the different available languages.

See Also

DROP FUNCTION, *GRANT*, *LOAD*, *REVOKE*, createlang, *PostgreSQL Programmer’s Guide*

CREATE GROUP

Name

`CREATE GROUP` — define a new user group

Synopsis

```
CREATE GROUP name [ [ WITH ] option [ ... ] ]

where option can be:

     SYSID gid
   | USER  username [, ...]
```

Inputs

`name`

The name of the group.

`gid`

The `SYSID` clause can be used to choose the PostgreSQL group id of the new group. It is not necessary to do so, however.

If this is not specified, the highest assigned group id plus one, starting at 1, will be used as default.

`username`

A list of users to include in the group. The users must already exist.

Outputs

`CREATE GROUP`

Message returned if the command completes successfully.

Description

CREATE GROUP will create a new group in the database installation. Refer to the *Administrator's Guide* for information about using groups for authentication. You must be a database superuser to use this command.

Use *ALTER GROUP* to change a group's membership, and *DROP GROUP* to remove a group.

Usage

Create an empty group:

```
CREATE GROUP staff;
```

Create a group with members:

```
CREATE GROUP marketing WITH USER jonathan, david;
```

Compatibility

SQL92

There is no `CREATE GROUP` statement in SQL92. Roles are similar in concept to groups.

CREATE INDEX

Name

CREATE INDEX — define a new index

Synopsis

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

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 data which would result in duplicate entries will generate an error.

index_name

The name of the index to be created. No schema name can be included here; the index is always created in the same schema as its parent table.

table

The name (possibly schema-qualified) of the table to be indexed.

acc_method

The name of the access method to be used for the index. The default access method is `BTREE`. PostgreSQL provides four access methods for indexes:

`BTREE`

an implementation of Lehman-Yao high-concurrency B-trees.

`RTREE`

implements standard R-trees using Guttman's quadratic split algorithm.

`HASH`

an implementation of Litwin's linear hashing.

`GIST`

Generalized Index Search Trees.

column

The name of a column of the table.

ops_name

An associated operator class. See below for details.

func_name

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

predicate

Defines the constraint expression for a partial index.

Outputs

`CREATE INDEX`

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 field(s) for the index are specified as column names. Multiple fields can be specified if the index access method supports multicolumn indexes.

In the second syntax shown above, an index is defined on the result of a user-specified function *func_name* applied to one or more columns of a single table. 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. For example, a functional index on `upper(col)` would allow the clause `WHERE upper(col) = 'JIM'` to use an index.

PostgreSQL provides B-tree, R-tree, hash, and GiST access methods for indexes. The B-tree access method is an implementation of Lehman-Yao high-concurrency B-trees. The R-tree access method implements standard R-trees using Guttman's quadratic split algorithm. The hash access method is an implementation of Litwin's linear hashing. We mention the algorithms used solely to indicate that all of these access methods are fully dynamic and do not have to be optimized periodically (as is the case with, for example, static hash access methods).

When the `WHERE` clause is present, a *partial index* is created. A partial index is an index that contains entries for only a portion of a table, usually a portion that is somehow more interesting than the rest of the table. For example, if you have a table that contains both billed and unbilled orders where the unbilled orders take up a small fraction of the total table and yet that is an often used section, you can improve performance by creating an index on just that portion. Another possible application is to use `WHERE` with `UNIQUE` to enforce uniqueness over a subset of a table.

The expression used in the `WHERE` clause may refer only to columns of the underlying table (but it can use all columns, not only the one(s) being indexed). Presently, subqueries and aggregate expressions are also forbidden in `WHERE`.

All functions and operators used in an index definition must be *immutable*, that is, their results must depend only on their input arguments and never on any outside influence (such as the contents of another table or the current time). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index, remember to mark the function immutable when you create it.

Use *DROP INDEX* to remove an index.

Notes

The PostgreSQL query optimizer will consider using a B-tree index whenever an indexed attribute is involved in a comparison using one of: <, <=, =, >=, >

The PostgreSQL query optimizer will consider using an R-tree index whenever an indexed attribute is involved in a comparison using one of: <<, &<, &>, >>, @, ~=, &&

The PostgreSQL query optimizer will consider using a hash index whenever an indexed attribute is involved in a comparison using the = operator.

Testing has shown PostgreSQL's hash indexes to be similar or slower than B-tree indexes, and the index size and build time for hash indexes is much worse. Hash indexes also suffer poor performance under high concurrency. For these reasons, hash index use is discouraged.

Currently, only the B-tree and gist access methods support multicolumn indexes. Up to 32 keys may be specified by default (this limit can be altered when building PostgreSQL). Only B-tree currently supports unique indexes.

An *operator class* can be specified for each column of an index. The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on four-byte integers would use the `int4_ops` class; this operator class includes comparison functions for four-byte integers. In practice the default operator class for the field's data type is usually sufficient. The main point of having operator classes is that for some data types, there could be more than one meaningful ordering. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index. There are also some operator classes with special purposes:

- The operator classes `box_ops` and `bigbox_ops` both support R-tree indexes on the `box` data type. The difference between them is that `bigbox_ops` scales box coordinates down, to avoid floating-point exceptions from doing multiplication, addition, and subtraction on very large floating-point coordinates. (Note: this was true some time ago, but currently the two operator classes both use floating point and are effectively identical.)

The following query shows all defined operator classes:

```
SELECT am.amname AS acc_method,
       opc.opcname AS ops_name
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcamid = am.oid
    ORDER BY acc_method, ops_name;
```

Usage

To create a B-tree index on the field `title` in the table `films`:

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

Compatibility

SQL92

CREATE INDEX is a PostgreSQL language extension.

There is no `CREATE INDEX` command in SQL92.

CREATE LANGUAGE

Name

`CREATE LANGUAGE` — define a new procedural language

Synopsis

```
CREATE [ TRUSTED ] [ PROCEDURAL ] LANGUAGE langname
    HANDLER call_handler [ VALIDATOR valfunction ]
```

Description

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

`CREATE LANGUAGE` effectively associates the language name with a call handler that is responsible for executing functions written in the language. Refer to the *Programmer's Guide* for more information about language call handlers.

Note that procedural languages are local to individual databases. To make a language available in all databases by default, it should be installed into the `template1` database.

Parameters

`TRUSTED`

> `TRUSTED` specifies that the call handler for the language is safe, that is, it does not offer an unprivileged user any functionality to bypass access restrictions. If this keyword is omitted when registering the language, only users with the PostgreSQL superuser privilege can use this language to create new functions.

`PROCEDURAL`

> This is a noise word.

`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 PostgreSQL.
>
> For backward compatibility, the name may be enclosed by single quotes.

`HANDLER` *`call_handler`*

> *`call_handler`* is the name of a previously registered function that will be called to execute the procedural language functions. The call handler for a procedural language must be written in a compiled language such as C with version 1 call convention and registered with PostgreSQL as a function taking no arguments and returning the `language_handler` type, a placeholder type that is simply used to identify the function as a call handler.

`VALIDATOR` *`valfunction`*

> *`valfunction`* is the name of a previously registered function that will be called when a new function in the language is created, to validate the new function. If no validator function is spec-

ified, then a new function will not be checked when it is created. The validator function must take one argument of type `oid`, which will be the OID of the to-be-created function, and will typically return `void`.

A validator function would typically inspect the function body for syntactical correctness, but it can also look at other properties of the function, for example if the language cannot handle certain argument types. To signal an error, the validator function should use the `elog()` function. The return value of the function is ignored.

Diagnostics

```
CREATE LANGUAGE
```

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.

Notes

This command normally should not be executed directly by users. For the procedural languages supplied in the PostgreSQL distribution, the createlang script should be used, which will also install the correct call handler. (`createlang` will call `CREATE LANGUAGE` internally.)

In PostgreSQL versions before 7.3, it was necessary to declare handler functions as returning the placeholder type `opaque`, rather than `language_handler`. To support loading of old dump files, `CREATE LANGUAGE` will accept a function declared as returning `opaque`, but it will issue a NOTICE and change the function’s declared return type to `language_handler`.

Use the *CREATE FUNCTION* command to create a new function.

Use *DROP LANGUAGE*, or better yet the droplang script, to drop procedural languages.

The system catalog `pg_language` records information about the currently installed procedural languages.

```
        Table "pg_language"
   Attribute   |   Type    | Modifier
---------------+-----------+----------
 lanname       | name      |
 lanispl       | boolean   |
 lanpltrusted  | boolean   |
 lanplcallfoid | oid       |
 lanvalidator  | oid       |
     lanacl        | aclitem[] |

       lanname   | lanispl | lanpltrusted | lanplcallfoid | lanvalidator | lanacl
    -------------+---------+-------------+---------------+--------------+-
-------
     internal    | f       | f           |             0 |         2246 |
     c           | f       | f           |             0 |         2247 |
     sql         | f       | t           |             0 |         2248 | {=U}
```

At present, with the exception of the permissions, the definition of a procedural language cannot be changed once it has been created.

To be able to use a procedural language, a user must be granted the `USAGE` privilege. The `createlang` program automatically grants permissions to everyone if the language is known to be trusted.

Examples

The following two commands executed in sequence will register a new procedural language and the associated call handler.

```
CREATE FUNCTION plsample_call_handler () RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

Compatibility

`CREATE LANGUAGE` is a PostgreSQL extension.

History

The `CREATE LANGUAGE` command first appeared in PostgreSQL 6.3.

See Also

createlang, *CREATE FUNCTION*, droplang, *DROP LANGUAGE*, *GRANT*, *REVOKE*, *PostgreSQL Programmer's Guide*

CREATE OPERATOR

Name

`CREATE OPERATOR` — define a new operator

Synopsis

```
CREATE OPERATOR name ( PROCEDURE = func_name
     [, LEFTARG = lefttype
     ] [, RIGHTARG = righttype ]
     [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
     [, RESTRICT = res_proc ] [, JOIN = join_proc ]
     [, HASHES ] [, MERGES ]
     [, SORT1 = left_sort_op ] [, SORT2 = right_sort_op ]
     [, LTCMP = less_than_op ] [, GTCMP = greater_than_op ] )
```

Inputs

name

The operator to be defined. See below for allowable characters. The name may be schema-qualified, for example `CREATE OPERATOR myschema.+ (...)`.

func_name

The function used to implement this operator.

lefttype

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

righttype

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

com_op

The commutator of this operator.

neg_op

The negator of this operator.

res_proc

The restriction selectivity estimator function for this operator.

join_proc

The join selectivity estimator function for this operator.

HASHES

Indicates this operator can support a hash join.

MERGES

Indicates this operator can support a merge join.

`left_sort_op`

If this operator can support a merge join, the less-than operator that sorts the left-hand data type of this operator.

`right_sort_op`

If this operator can support a merge join, the less-than operator that sorts the right-hand data type of this operator.

`less_than_op`

If this operator can support a merge join, the less-than operator that compares the input data types of this operator.

`greater_than_op`

If this operator can support a merge join, the greater-than operator that compares the input data types of this operator.

Outputs

`CREATE OPERATOR`

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.

If a schema name is given then the operator is created in the specified schema. Otherwise it is created in the current schema (the one at the front of the search path; see `CURRENT_SCHEMA()`).

Two operators in the same schema can have the same name if they operate on different data types. This is called *overloading*. The system will attempt to pick the intended operator based on the actual input data types when there is ambiguity.

The operator *`name`* is a sequence of up to `NAMEDATALEN`-1 (63 by default) characters from the following list:

+ - * / < > = ~ ! @ # % ^ & | ‘ ? $

There are a few restrictions on your choice of name:

- `$` cannot be defined as a single-character operator, although it can be part of a multicharacter operator name.

- `--` and `/*` cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
- A multicharacter operator name cannot end in `+` or `-`, unless the name also contains at least one of these characters:

  ~ ! @ # % ^ & | ' ? $

  For example, `@-` is an allowed operator name, but `*-` is not. This restriction allows PostgreSQL to parse SQL-compliant queries without requiring spaces between tokens.

> **Note:** When working with non-SQL-standard operator names, you will usually need to separate adjacent operators with spaces to avoid ambiguity. For example, if you have defined a left-unary operator named `@`, you cannot write `X*@Y`; you must write `X* @Y` to ensure that PostgreSQL reads it as two operator names not one.

The operator `!=` is mapped to <> on input, so these two names are always 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.

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) of the indicated types.

The commutator operator should be identified if one exists, so that PostgreSQL can reverse the order of the operands if it wishes. For example, the operator area-less-than, <<<, would probably have a commutator operator, area-greater-than, >>>. Hence, the query optimizer could freely convert:

```
box '((0,0), (1,1))'  >>> MYBOXES.description
```

to

```
MYBOXES.description <<< box '((0,0), (1,1))'
```

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

Similarly, if there is a negator operator then it should be identified. Suppose that an operator, area-equal, ===, exists, as well as an area not equal, !==. The negator link allows the query optimizer to simplify

```
NOT MYBOXES.description === box '((0,0), (1,1))'
```

to

```
MYBOXES.description !== box '((0,0), (1,1))'
```

If a commutator operator name is supplied, PostgreSQL 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 newly created 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. (As of PostgreSQL `6.5`, it also works to just have both operators refer to each other.)

The `HASHES`, `MERGES`, `SORT1`, `SORT2`, `LTCMP`, and `GTCMP` options are present to support the query optimizer in performing joins. PostgreSQL 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 . In addition, PostgreSQL can use a hash-join algorithm ; however, it must know whether this strategy is applicable. The current hash-join algorithm is only correct for operators that represent equality tests; furthermore, equality of the data type must mean bitwise equality of the representation of the type. (For example, a data type that contains unused bits that don't matter for equality tests could not be hash-joined.) The `HASHES` flag indicates to the query optimizer that a hash join may safely be used with this operator.

Similarly, the `MERGES` flag indicates whether merge-sort is a usable join strategy for this operator. A merge join requires that the two input data types have consistent orderings, and that the merge-join operator behave like equality with respect to that ordering. For example, it is possible to merge-join equality between an integer and a float variable by sorting both inputs in ordinary numeric order. Execution of a merge join requires that the system be able to identify four operators related to the merge-join equality operator: less-than comparison for the left input data type, less-than comparison for the right input data type, less-than comparison between the two data types, and greater-than comparison between the two data types. It is possible to specify these by name, as the `SORT1`, `SORT2`, `LTCMP`, and `GTCMP` options respectively. The system will fill in the default names <, <, <, > respectively if any of these are omitted when `MERGES` is specified. Also, `MERGES` will be assumed to be implied if any of these four operator options appear.

If other join strategies are found to be practical, PostgreSQL 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 `RESTRICT` and `JOIN` options assist the query optimizer in estimating result sizes. If a clause of the form:

```
myboxes.description <<< box '((0,0), (1,1))'
```

is present in the qualification, then PostgreSQL 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 `CREATE FUNCTION`) which accepts arguments of the correct data types 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 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 tables involved to compute the expected result size.

The difference between the function

```
my_procedure_1 (MYBOXES.description, box '((0,0), (1,1))')
```

and the operator

```
MYBOXES.description === box '((0,0), (1,1))'
```

is that PostgreSQL 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.

To give a schema-qualified operator name in *`com_op`* or the other optional arguments, use the `OPERATOR()` syntax, for example

```
   COMMUTATOR = OPERATOR(myschema.===) ,
```

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,
   JOIN = area_join_procedure,
   HASHES,
   SORT1 = <<<,
   SORT2 = <<<
   -- Since sort operators were given, MERGES is implied.
   -- LTCMP and GTCMP are assumed to be < and > respectively
);
```

Compatibility

SQL92

`CREATE OPERATOR` is a PostgreSQL extension. There is no `CREATE OPERATOR` statement in SQL92.

CREATE OPERATOR CLASS

Name

CREATE OPERATOR CLASS — define a new operator class for indexes

Synopsis

```
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type USING access_method AS
  {  OPERATOR strategy_number operator_id [ ( type, type ) ] [ RECHECK ]
   | FUNCTION support_number func_name ( parameter_types )
   | STORAGE storage_type
  } [, ... ]
```

Inputs

name

The name of the operator class to be created. The name may be schema-qualified.

DEFAULT

If present, the operator class will become the default index operator class for its data type. At most one operator class can be the default for a specific data type and access method.

data_type

The column data type that this operator class is for.

access_method

The name of the index access method this operator class is for.

strategy_number

The index access method's strategy number for an operator associated with the operator class.

operator_id

The identifier (optionally schema-qualified) of an operator associated with the operator class.

type

The input data type(s) of an operator, or NONE to signify a left-unary or right-unary operator. The input data types may be omitted in the normal case where they are the same as the operator class's data type.

RECHECK

If present, the index is "lossy" for this operator, and so the tuples retrieved using the index must be rechecked to verify that they actually satisfy the qualification clause involving this operator.

support_number

The index access method's support procedure number for a function associated with the operator class.

func_name

The name (optionally schema-qualified) of a function that is an index access method support procedure for the operator class.

`parameter_types`

The parameter data type(s) of the function.

`storage_type`

The data type actually stored in the index. Normally this is the same as the column data type, but some index access methods (only GIST at this writing) allow it to be different. The `STORAGE` clause must be omitted unless the index access method allows a different type to be used.

Outputs

`CREATE OPERATOR CLASS`

Message returned if the operator class is successfully created.

Description

`CREATE OPERATOR CLASS` defines a new operator class, *`name`*.

An operator class defines how a particular data type can be used with an index. The operator class specifies that certain operators will fill particular roles or “strategies” for this data type and this access method. The operator class also specifies the support procedures to be used by the index access method when the operator class is selected for an index column. All the operators and functions used by an operator class must be defined before the operator class is created.

If a schema name is given then the operator class is created in the specified schema. Otherwise it is created in the current schema (the one at the front of the search path; see `CURRENT_SCHEMA()`). Two operator classes in the same schema can have the same name only if they are for different index access methods.

The user who defines an operator class becomes its owner. Presently, the creating user must be a superuser. (This restriction is made because an erroneous operator class definition could confuse or even crash the server.)

`CREATE OPERATOR CLASS` does not presently check whether the class definition includes all the operators and functions required by the index access method. It is the user’s responsibility to define a valid operator class.

Refer to the chapter on interfacing extensions to indexes in the *PostgreSQL Programmer’s Guide* for further information.

Notes

Refer to *DROP OPERATOR CLASS* to delete user-defined operator classes from a database.

Usage

The following example command defines a GiST index operator class for data type `_int4` (array of int4). See `contrib/intarray/` for the complete example.

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR        3       &&,
        OPERATOR        6       =       RECHECK,
        OPERATOR        7       @,
        OPERATOR        8       ~,
        OPERATOR        20      @@ (_int4, query_int),
        FUNCTION        1       g_int_consistent (internal, _int4, int4),
        FUNCTION        2       g_int_union (bytea, internal),
        FUNCTION        3       g_int_compress (internal),
        FUNCTION        4       g_int_decompress (internal),
        FUNCTION        5       g_int_penalty (internal, internal, internal),
        FUNCTION        6       g_int_picksplit (internal, internal),
        FUNCTION        7       g_int_same (_int4, _int4, internal);
```

The `OPERATOR`, `FUNCTION`, and `STORAGE` clauses may appear in any order.

Compatibility

SQL92

`CREATE OPERATOR CLASS` is a PostgreSQL extension. There is no `CREATE OPERATOR CLASS` statement in SQL92.

CREATE RULE

Name

`CREATE RULE` — define a new rewrite rule

Synopsis

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table [ WHERE condition ]
    DO [ INSTEAD ] action

where action can be:

NOTHING
| query
| ( query ; query ... )
```

Inputs

name

The name of a rule to create. This must be distinct from the name of any other rule for the same table.

event

Event is one of `SELECT`, `UPDATE`, `DELETE` or `INSERT`.

table

The name (optionally schema-qualified) of the table or view the rule applies to.

condition

Any SQL conditional expression (returning `boolean`). The condition expression may not refer to any tables except `new` and `old`, and may not contain aggregate functions.

query

The query or queries making up the *action* can be any SQL `SELECT`, `INSERT`, `UPDATE`, `DELETE`, or `NOTIFY` statement.

Within the *condition* and *action*, the special table names `new` and `old` may be used to refer to values in the referenced table. `new` is valid in ON INSERT and ON UPDATE rules to refer to the new row being inserted or updated. `old` is valid in ON UPDATE and ON DELETE rules to refer to the existing row being updated or deleted.

Outputs

`CREATE RULE`

Message returned if the rule is successfully created.

Description

`CREATE RULE` defines a new rule applying to a specified table or view. `CREATE OR REPLACE RULE` will either create a new rule, or replace an existing rule of the same name for the same table.

The PostgreSQL *rule system* allows one to define an alternate action to be performed on inserts, updates, or deletions from database tables. Rules are used to implement table views as well.

The semantics of a rule is that at the time an individual instance (row) is accessed, inserted, updated, or deleted, there is an old instance (for selects, updates and deletes) and a new instance (for inserts and updates). All the rules for the given event type and the given target table are examined successively (in order by name). If the *`condition`* specified in the WHERE clause (if any) is true, the *`action`* part of the rule is executed. The *`action`* is done instead of the original query if INSTEAD is specified; otherwise it is done after the original query in the case of ON INSERT, or before the original query in the case of ON UPDATE or ON DELETE. Within both the *`condition`* and *`action`*, values from fields in the old instance and/or the new instance are substituted for `old.`*`attribute-name`* and `new.`*`attribute-name`*.

The *`action`* part of the rule can consist of one or more queries. To write multiple queries, surround them with parentheses. Such queries will be performed in the specified order. The *`action`* can also be NOTHING indicating no action. Thus, a DO INSTEAD NOTHING rule suppresses the original query from executing (when its condition is true); a DO NOTHING rule is useless.

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

It is important to realize that a rule is really a query transformation mechanism, or query macro. The entire query is processed to convert it into a series of queries that include the rule actions. This occurs before evaluation of the query starts. So, conditional rules are handled by adding the rule condition to the WHERE clause of the action(s) derived from the rule. The above description of a rule as an operation that executes for each row is thus somewhat misleading. If you actually want an operation that fires independently for each physical row, you probably want to use a trigger not a rule. Rules are most useful for situations that call for transforming entire queries independently of the specific data being handled.

Rules and Views

Presently, ON SELECT rules must be unconditional INSTEAD rules and must have actions that consist of a single SELECT query. Thus, an ON SELECT rule effectively turns the table into a view, whose visible contents are the rows returned by the rule's SELECT query rather than whatever had been stored in the table (if anything). It is considered better style to write a CREATE VIEW command than to create a real table and define an ON SELECT rule for it.

CREATE VIEW creates a dummy table (with no underlying storage) and associates an ON SELECT rule with it. The system will not allow updates to the view, since it knows there is no real table there. You can create the illusion of an updatable view by defining ON INSERT, ON UPDATE, and ON DELETE rules (or any subset of those that's sufficient for your purposes) to replace update actions on the view with appropriate updates on other tables.

There is a catch if you try to use conditional rules for view updates: there *must* be an unconditional INSTEAD rule for each action you wish to allow on the view. If the rule is conditional, or is not

INSTEAD, then the system will still reject attempts to perform the update action, because it thinks it might end up trying to perform the action on the dummy table in some cases. If you want to handle all the useful cases in conditional rules, you can; just add an unconditional DO INSTEAD NOTHING rule to ensure that the system understands it will never be called on to update the dummy table. Then make the conditional rules non-INSTEAD; in the cases where they fire, they add to the default INSTEAD NOTHING action.

Notes

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

It is very important to take care to avoid circular rules. For example, though each of the following two rule definitions are accepted by PostgreSQL, the select command will cause PostgreSQL to report an error because the query cycled too many times:

```
CREATE RULE "_RETURN" AS
    ON SELECT TO emp
    DO INSTEAD
 SELECT * FROM toyemp;

CREATE RULE "_RETURN" AS
    ON SELECT TO toyemp
    DO INSTEAD
 SELECT * FROM emp;
```

This attempt to select from `EMP` will cause PostgreSQL to issue an error because the queries cycled too many times:

```
SELECT * FROM emp;
```

Presently, if a rule contains a NOTIFY query, the NOTIFY will be executed unconditionally --- that is, the NOTIFY will be issued even if there are not any rows that the rule should apply to. For example, in

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO NOTIFY mytable;

UPDATE mytable SET name = ’foo’ WHERE id = 42;
```

one NOTIFY event will be sent during the UPDATE, whether or not there are any rows with id = 42. This is an implementation restriction that may be fixed in future releases.

Compatibility

SQL92

`CREATE RULE` is a PostgreSQL language extension. There is no `CREATE RULE` statement in SQL92.

CREATE SCHEMA

Name

`CREATE SCHEMA` — define a new schema

Synopsis

```
CREATE SCHEMA schemaname [ AUTHORIZATION username ] [ schema_element [ ... ] ]
CREATE SCHEMA AUTHORIZATION username [ schema_element [ ... ] ]
```

Inputs

`schemaname`

The name of a schema to be created. If this is omitted, the user name is used as the schema name.

`username`

The name of the user who will own the schema. If omitted, defaults to the user executing the command. Only superusers may create schemas owned by users other than themselves.

`schema_element`

An SQL statement defining an object to be created within the schema. Currently, only `CREATE TABLE`, `CREATE VIEW`, and `GRANT` are accepted as clauses within `CREATE SCHEMA`. Other kinds of objects may be created in separate commands after the schema is created.

Outputs

`CREATE SCHEMA`

Message returned if the command is successful.

`ERROR: namespace "`*`schemaname`*`" already exists`

If the schema specified already exists.

Description

`CREATE SCHEMA` will enter a new schema into the current database. The schema name must be distinct from the name of any existing schema in the current database.

A schema is essentially a namespace: it contains named objects (tables, data types, functions, and operators) whose names may duplicate those of other objects existing in other schemas. Named objects are accessed either by “qualifying” their names with the schema name as a prefix, or by setting a search path that includes the desired schema(s). Unqualified objects are created in the current schema (the one at the front of the search path; see `CURRENT_SCHEMA()`).

Optionally, `CREATE SCHEMA` can include subcommands to create objects within the new schema. The subcommands are treated essentially the same as separate commands issued after creating the schema, except that if the `AUTHORIZATION` clause is used, all the created objects will be owned by that user.

Notes

To create a schema, the invoking user must have `CREATE` privilege for the current database. (Of course, superusers bypass this check.)

Use `DROP SCHEMA` to remove a schema.

Examples

Create a schema:

```
CREATE SCHEMA myschema;
```

Create a schema for user `joe` --- the schema will also be named `joe`:

```
CREATE SCHEMA AUTHORIZATION joe;
```

Create a schema and create a table and view within it:

```
CREATE SCHEMA hollywood
    CREATE TABLE films (title text, release date, awards text[])
    CREATE VIEW winners AS
        SELECT title, release FROM films WHERE awards IS NOT NULL;
```

Notice that the individual subcommands do not end with semicolons.

The following is an equivalent way of accomplishing the same result:

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (title text, release date, awards text[]);
CREATE VIEW hollywood.winners AS
    SELECT title, release FROM hollywood.films WHERE awards IS NOT NULL;
```

Compatibility

SQL92

SQL92 allows a `DEFAULT CHARACTER SET` clause in `CREATE SCHEMA`, as well as more subcommand types than are presently accepted by PostgreSQL.

SQL92 specifies that the subcommands in `CREATE SCHEMA` may appear in any order. The present PostgreSQL implementation does not handle all cases of forward references in subcommands; it may sometimes be necessary to reorder the subcommands to avoid forward references.

In SQL92, the owner of a schema always owns all objects within it. PostgreSQL allows schemas to contain objects owned by users other than the schema owner. This can happen only if the schema owner grants `CREATE` rights on his schema to someone else.

CREATE SEQUENCE

Name

`CREATE SEQUENCE` — define a new sequence generator

Synopsis

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

Inputs

TEMPORARY or TEMP

> If specified, the sequence object is created only for this session, and is automatically dropped on session exit. Existing permanent sequences with the same name are not visible (in this session) while the temporary sequence exists, unless they are referenced with schema-qualified names.

seqname

> The name (optionally schema-qualified) 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 -2^63-1 for ascending and descending sequences, respectively.

maxvalue

> The optional clause `MAXVALUE` *maxvalue* determines the maximum value for the sequence. The defaults are 2^63-1 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 wrap around 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 the *minvalue* or *maxvalue*, respectively. Without CYCLE, after the limit is reached `nextval` calls will return an error.

Outputs

`CREATE SEQUENCE`

Message returned if the command is successful.

`ERROR: Relation '`*`seqname`*`' already exists`

If the sequence specified already exists.

`ERROR: DefineSequence: MINVALUE (`*`start`*`) can't be >= MAXVALUE (`*`max`*`)`

If the specified starting value is out of range.

`ERROR: DefineSequence: START value (`*`start`*`) can't be < MINVALUE (`*`min`*`)`

If the specified starting value is out of range.

`ERROR: DefineSequence: MINVALUE (`*`min`*`) can't be >= MAXVALUE (`*`max`*`)`

If the minimum and maximum values are inconsistent.

Description

`CREATE SEQUENCE` will enter a new sequence number generator into the current database. This involves creating and initializing a new single-row table with the name *`seqname`*. The generator will be owned by the user issuing the command.

If a schema name is given then the sequence is created in the specified schema. Otherwise it is created in the current schema (the one at the front of the search path; see `CURRENT_SCHEMA()`). TEMP sequences exist in a special schema, so a schema name may not be given when creating a TEMP sequence. The sequence name must be distinct from the name of any other sequence, table, index, or view in the same schema.

After a sequence is created, you use the functions `nextval`, `currval` and `setval` to operate on the sequence. These functions are documented in the *User's Guide*.

Although you cannot update a sequence directly, you can use a query like

```
SELECT * FROM seqname;
```

to examine the parameters and current state of a sequence. In particular, the `last_value` field of the sequence shows the last value allocated by any backend process. (Of course, this value may be obsolete by the time it's printed, if other processes are actively doing `nextval` calls.)

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 and 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, any numbers allocated but not used within a session will be lost when that session ends. 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`. Another consideration is that a `setval` executed on such a sequence will not be noticed by other backends until they have used up any preallocated values they have cached.

Notes

Use `DROP SEQUENCE` to remove a sequence.

Sequences are based on `bigint` arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807). On some older platforms, there may be no compiler support for eight-byte integers, in which case sequences use regular `integer` arithmetic (range -2147483648 to +2147483647).

When *cache* is greater than one, each backend uses its own cache to store preallocated numbers. Numbers that are cached but not used in the current session will be lost, resulting in "holes" in the sequence.

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’);
```

Update the sequence value after a COPY FROM:

```
BEGIN;
    COPY distributors FROM ’input_file’;
    SELECT setval(’serial’, max(id)) FROM distributors;
END;
```

Compatibility

SQL92

`CREATE SEQUENCE` is a PostgreSQL language extension. There is no `CREATE SEQUENCE` statement in SQL92.

CREATE TABLE

Name

`CREATE TABLE` — define a new table

Synopsis

```
    CREATE [ [ LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name (
        { column_name data_type [ DEFAULT default_expr ] [ column_constraint [, ... ] ]
        | table_constraint }  [, ... ]
    )
    [ INHERITS ( parent_table [, ... ] ) ]
    [ WITH OIDS | WITHOUT OIDS ]

    where column_constraint is:

    [ CONSTRAINT constraint_name ]
    { NOT NULL | NULL | UNIQUE | PRIMARY KEY |
      CHECK (expression) |
      REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL ]
        [ ON DELETE action ] [ ON UPDATE action ] }
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDI-
ATE ]

    and table_constraint is:

    [ CONSTRAINT constraint_name ]
    { UNIQUE ( column_name [, ... ] ) |
      PRIMARY KEY ( column_name [, ... ] ) |
      CHECK ( expression ) |
      FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
        [ MATCH FULL | MATCH PARTIAL ] [ ON DELETE action ] [ ON UPDATE ac-
tion ] }
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDI-
ATE ]
```

Description

`CREATE TABLE` will create a new, initially empty table in the current database. The table will be owned by the user issuing the command.

If a schema name is given (for example, `CREATE TABLE myschema.mytable ...`) then the table is created in the specified schema. Otherwise it is created in the current schema (the one at the front of the search path; see `CURRENT_SCHEMA()`). TEMP tables exist in a special schema, so a schema name may not be given when creating a TEMP table. The table name must be distinct from the name of any other table, sequence, index, or view in the same schema.

`CREATE TABLE` also automatically creates a data type that represents the tuple type (structure type) corresponding to one row of the table. Therefore, tables cannot have the same name as any existing data type in the same schema.

A table cannot have more than 1600 columns. (In practice, the effective limit is lower because of tuple-length constraints).

The optional constraint clauses specify constraints (or tests) that new or updated rows must satisfy for an insert or update operation to succeed. 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 table.

There are two ways to define constraints: table constraints and column constraints. A column constraint is defined as part of a column definition. A table constraint definition is not tied to a particular column, and it can encompass more than one column. Every column constraint can also be written as a table constraint; a column constraint is only a notational convenience if the constraint only affects one column.

Parameters

`[LOCAL] TEMPORARY` or `[LOCAL] TEMP`

If specified, the table is created as a temporary table. Temporary tables are automatically dropped at the end of a session. Existing permanent tables with the same name are not visible to the current session while the temporary table exists, unless they are referenced with schema-qualified names. Any indexes created on a temporary table are automatically temporary as well.

The `LOCAL` word is optional. But see under *Compatibility*.

`table_name`

The name (optionally schema-qualified) of the table to be created.

`column_name`

The name of a column to be created in the new table.

`data_type`

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

`DEFAULT` *`default_expr`*

The `DEFAULT` clause assigns a default data value for the column whose column definition it appears within. The value is any variable-free expression (subselects and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column.

The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is NULL.

`INHERITS (` *`parent_table`* `[, ... ] )`

The optional `INHERITS` clause specifies a list of tables from which the new table automatically inherits all columns. If the same column name exists in more than one parent table, an error is reported unless the data types of the columns match in each of the parent tables. If there is no conflict, then the duplicate columns are merged to form a single column in the new table. If the column name list of the new table contains a column that is also inherited, the data type must likewise match the inherited column(s), and the column definitions are merged into one. However, inherited and new column declarations of the same name need not specify identical constraints: all constraints provided from any declaration are merged together and all are applied to the new table. If the new table explicitly specifies a default value for the column, this default overrides any defaults from inherited declarations of the column. Otherwise, any parents that specify default values for the column must all specify the same default, or an error will be reported.

`WITH OIDS` or `WITHOUT OIDS`

This optional clause specifies whether rows of the new table should have OIDs (object identifiers) assigned to them. The default is to have OIDs. (If the new table inherits from any tables that have OIDs, then `WITH OIDS` is forced even if the command says `WITHOUT OIDS`.)

Specifying `WITHOUT OIDS` allows the user to suppress generation of OIDs for rows of a table. This may be worthwhile for large tables, since it will reduce OID consumption and thereby postpone wraparound of the 32-bit OID counter. Once the counter wraps around, uniqueness of OIDs can no longer be assumed, which considerably reduces their usefulness.

`CONSTRAINT` *`constraint_name`*

An optional name for a column or table constraint. If not specified, the system generates a name.

`NOT NULL`

The column is not allowed to contain NULL values.

`NULL`

The column is allowed to contain NULL values. This is the default.

This clause is only available for compatibility with non-standard SQL databases. Its use is discouraged in new applications.

`UNIQUE` (column constraint)
`UNIQUE (` *`column_name`* `[, ... ] )` (table constraint)

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.

For the purpose of a unique constraint, NULL values are not considered equal.

Each unique table constraint must name a set of columns that is different from the set of columns named by any other unique or primary key constraint defined for the table. (Otherwise it would just be the same constraint listed twice.)

`PRIMARY KEY` (column constraint)
`PRIMARY KEY (` *`column_name`* `[, ... ] )` (table constraint)

The primary key constraint specifies that a column or columns of a table may contain only unique (non-duplicate), non-NULL values. Technically, `PRIMARY KEY` is merely a combination of `UNIQUE` and `NOT NULL`, but identifying a set of columns as primary key also provides metadata about the design of the schema, as a primary key implies that other tables may rely on this set of columns as a unique identifier for rows.

Only one primary key can be specified for a table, whether as a column constraint or a table constraint.

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.

`CHECK (`*`expression`*`)`

`CHECK` clauses specify integrity constraints or tests which new or updated rows must satisfy for an insert or update operation to succeed. Each constraint must be an expression producing a Boolean result. A condition appearing within a column definition should reference that column's value only, while a condition appearing as a table constraint may reference multiple columns.

Currently, `CHECK` expressions cannot contain subselects nor refer to variables other than columns of the current row.

`REFERENCES` *`reftable`* `[ (` *`refcolumn`* `) ] [ MATCH` *`matchtype`* `] [ ON DELETE` *`action`* `] [ ON UPDATE` *`action`* `]` (column constraint)
`FOREIGN KEY (` *`column`* `[, ... ] ) REFERENCES` *`reftable`* `[ (` *`refcolumn`* `[, ... ] ) ] [ MATCH` *`matchtype`* `] [ ON DELETE` *`action`* `] [ ON UPDATE` *`action`* `]` (table constraint)

The `REFERENCES` column constraint specifies that a group of one or more columns of the new table must only contain values which match against values in the referenced column(s) *`refcolumn`* of the referenced table *`reftable`*. If *`refcolumn`* is omitted, the primary key of the *`reftable`* is used. The referenced columns must be the columns of a unique or primary key constraint in the referenced table.

A value added to these columns is matched against the values of the referenced table and referenced columns using the given match type. There are three match types: `MATCH FULL`, `MATCH PARTIAL`, and a default match type if none is specified. `MATCH FULL` will not allow one column of a multicolumn foreign key to be NULL unless all foreign key columns are NULL. The default match type allows some foreign key columns to be NULL while other parts of the foreign key are not NULL. `MATCH PARTIAL` is not yet implemented.

In addition, when the data in the referenced columns is changed, certain actions are performed on the data in this table's columns. The `ON DELETE` clause specifies the action to do when a referenced row in the referenced table is being deleted. Likewise, the `ON UPDATE` clause specifies the action to perform when a referenced column in the referenced table is being updated to a new value. If the row is updated, but the referenced column is not actually changed, no action is done. There are the following possible actions for each clause:

`NO ACTION`

Produce an error indicating that the deletion or update would create a foreign key constraint violation. This is the default action.

`RESTRICT`

Same as `NO ACTION`.

`CASCADE`

Delete any rows referencing the deleted row, or update the value of the referencing column to the new value of the referenced column, respectively.

`SET NULL`

Set the referencing column values to NULL.

`SET DEFAULT`

Set the referencing column values to their default value.

If primary key column is updated frequently, it may be wise to add an index to the `REFERENCES` column so that `NO ACTION` and `CASCADE` actions associated with the `REFERENCES` column can be more efficiently performed.

`DEFERRABLE` or `NOT DEFERRABLE`

This controls whether the constraint can be deferred. A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable may be postponed until the end of the transaction (using the *SET CONSTRAINTS* command). `NOT`

`DEFERRABLE` is the default. Only foreign key constraints currently accept this clause. All other constraint types are not deferrable.

`INITIALLY IMMEDIATE` or `INITIALLY DEFERRED`

If a constraint is deferrable, this clause specifies the default time to check the constraint. If the constraint is `INITIALLY IMMEDIATE`, it is checked after each statement. This is the default. If the constraint is `INITIALLY DEFERRED`, it is checked only at the end of the transaction. The constraint check time can be altered with the *SET CONSTRAINTS* command.

Diagnostics

`CREATE TABLE`

Message returned if table is successfully created.

`ERROR`

Message returned if table creation failed. This is usually accompanied by some descriptive text, such as: `ERROR: Relation '`*`table`*`' already exists`, which occurs at run time if the table specified already exists in the database.

Notes

- Whenever an application makes use of OIDs to identify specific rows of a table, it is recommended to create a unique constraint on the `oid` column of that table, to ensure that OIDs in the table will indeed uniquely identify rows even after counter wraparound. Avoid assuming that OIDs are unique across tables; if you need a database-wide unique identifier, use the combination of `tableoid` and row OID for the purpose. (It is likely that future PostgreSQL releases will use a separate OID counter for each table, so that it will be *necessary*, not optional, to include `tableoid` to have a unique identifier database-wide.)

  Tip: The use of `WITHOUT OIDS` is not recommended for tables with no primary key, since without either an OID or a unique data key, it is difficult to identify specific rows.

- PostgreSQL automatically creates an index for each unique constraint and primary key constraint to enforce the uniqueness. Thus, it is not necessary to create an explicit index for primary key columns. (See *CREATE INDEX* for more information.)

- The SQL92 standard says that `CHECK` column constraints may only refer to the column they apply to; only `CHECK` table constraints may refer to multiple columns. PostgreSQL does not enforce this restriction; it treats column and table check constraints alike.
- Unique constraints and primary keys are not inherited in the current implementation. This makes the combination of inheritance and unique constraints rather dysfunctional.

Examples

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(3) 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(3),
    date_prod   DATE,
    kind        VARCHAR(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(5),
    title       VARCHAR(40),
    did         DECIMAL(3),
    date_prod   DATE,
    kind        VARCHAR(10),
    len         INTERVAL HOUR TO MINUTE,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

Define a primary key constraint for table `distributors`. The following two examples are equivalent, the first using the table constraint syntax, the second the column constraint notation.

```
CREATE TABLE distributors (
    did     DECIMAL(3),
    name    CHAR VARYING(40),
    PRIMARY KEY(did)
);

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

This assigns a literal constant default value for the column `name`, and arranges for the default value of column `did` to be generated by selecting the next value of a sequence object. The default value of `modtime` will be the time at which the row is inserted.

```
CREATE TABLE distributors (
    name      VARCHAR(40) DEFAULT ’luso films’,
    did       INTEGER DEFAULT NEXTVAL(’distributors_serial’),
    modtime   TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Define two `NOT NULL` column constraints on the table `distributors`, one of which is explicitly given a name:

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

Define a unique constraint for the `name` column:

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

The above is equivalent to the following specified as a table constraint:

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

Compatibility

The `CREATE TABLE` conforms to SQL92 Intermediate and to a subset of SQL99, with exceptions listed below and in the descriptions above.

Temporary Tables

In addition to the local temporary table, SQL92 also defines a `CREATE GLOBAL TEMPORARY TABLE` statement. Global temporary tables are also visible to other sessions.

For temporary tables, there is an optional `ON COMMIT` clause:

```
    CREATE { GLOBAL | LOCAL } TEMPORARY TABLE table ( ... ) [ ON COMMIT { DELETE | PRE-
SERVE } ROWS ]
```

The `ON COMMIT` clause specifies whether or not the temporary table should be emptied of rows whenever `COMMIT` is executed. If the `ON COMMIT` clause is omitted, SQL92 specifies that the default is `ON COMMIT DELETE ROWS`. However, the behavior of PostgreSQL is always like `ON COMMIT PRESERVE ROWS`.

`NULL` “Constraint”

The `NULL` “constraint” (actually a non-constraint) is a PostgreSQL extension to SQL92 that is included for compatibility with some other RDBMS (and for symmetry with the `NOT NULL` constraint). Since it is the default for any column, its presence is simply noise.

Assertions

An assertion is a special type of integrity constraint and shares the same namespace as other constraints. However, an assertion is not necessarily dependent on one particular table as constraints are, so SQL92 provides the `CREATE ASSERTION` statement as an alternate method for defining a constraint:

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

PostgreSQL does not implement assertions at present.

Inheritance

Multiple inheritance via the `INHERITS` clause is a PostgreSQL language extension. SQL99 (but not SQL92) defines single inheritance using a different syntax and different semantics. SQL99-style inheritance is not yet supported by PostgreSQL.

Object IDs

The PostgreSQL concept of OIDs is not standard.

See Also

ALTER TABLE, *DROP TABLE*

CREATE TABLE AS

Name

`CREATE TABLE AS` — create a new table from the results of a query

Synopsis

```
CREATE [ [ LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name [ (column_name [, ...] )
    AS query
```

Description

`CREATE TABLE AS` creates a table and fills it with data computed by a `SELECT` command. The table columns have the names and data types associated with the output columns of the `SELECT` (except that you can override the column names by giving an explicit list of new column names).

`CREATE TABLE AS` bears some resemblance to creating a view, but it is really quite different: it creates a new table and evaluates the query just once to fill the new table initially. The new table will not track subsequent changes to the source tables of the query. In contrast, a view re-evaluates its defining `SELECT` statement whenever it is queried.

Parameters

`[LOCAL] TEMPORARY` or `[LOCAL] TEMP`

If specified, the table is created as a temporary table. Refer to *CREATE TABLE* for details.

`table_name`

The name (optionally schema-qualified) of the table to be created.

`column_name`

The name of a column in the new table. Multiple column names can be specified using a comma-delimited list of column names. If column names are not provided, they are taken from the output column names of the query.

`query`

A query statement (that is, a `SELECT` command). Refer to *SELECT* for a description of the allowed syntax.

Diagnostics

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

Notes

This command is functionally equivalent to *SELECT INTO*, but it is preferred since it is less likely to be confused with other uses of the `SELECT ... INTO` syntax.

Compatibility

This command is modeled after an Oracle feature. There is no command with equivalent functionality in SQL92 or SQL99. However, a combination of `CREATE TABLE` and `INSERT ... SELECT` can accomplish the same thing with little more effort.

History

The `CREATE TABLE AS` command has been available since PostgreSQL 6.3.

See Also

CREATE TABLE, *CREATE VIEW*, *SELECT*, *SELECT INTO*

CREATE TRIGGER

Name

`CREATE TRIGGER` — define a new trigger

Synopsis

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

Inputs

`name`

The name to give the new trigger. This must be distinct from the name of any other trigger for the same table.

`event`

One of INSERT, DELETE or UPDATE.

`table`

The name (optionally schema-qualified) of the table the trigger is for.

`func`

A user-supplied function that is declared as taking no arguments and returning type `trigger`.

`arguments`

An optional comma-separated list of arguments to be provided to the function when the trigger is executed, along with the standard trigger data such as old and new tuple contents. The arguments are literal string constants. Simple names and numeric constants may be written here too, but they will all be converted to strings.

Outputs

`CREATE TRIGGER`

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 *`table`* and will execute the specified function *`func`*.

The trigger can be specified to fire either before 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.

If multiple triggers of the same kind are defined for the same event, they will be fired in alphabetical order by name.

`SELECT` does not modify any rows so you can not create `SELECT` triggers. Rules and views are more appropriate in such cases.

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

Notes

To create a trigger on a table, the user must have the `TRIGGER` privilege on the table.

In PostgreSQL versions before 7.3, it was necessary to declare trigger functions as returning the placeholder type `opaque`, rather than `trigger`. To support loading of old dump files, `CREATE TRIGGER` will accept a function declared as returning `opaque`, but it will issue a NOTICE and change the function's declared return type to `trigger`.

As of the current release, `STATEMENT` triggers are not implemented.

Refer to the *DROP TRIGGER* command for information on how to remove triggers.

Examples

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');
```

The second example can 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
    );
```

Compatibility

SQL92

There is no `CREATE TRIGGER` statement in SQL92.

SQL99

The `CREATE TRIGGER` statement in PostgreSQL implements a subset of the SQL99 standard. The following functionality is missing:

- SQL99 allows triggers to fire on updates to specific columns (e.g., `AFTER UPDATE OF col1, col2`).
- SQL99 allows you to define aliases for the "old" and "new" rows or tables for use in the definition of the triggered action (e.g., `CREATE TRIGGER ... ON tablename REFERENCING OLD ROW AS somename NEW ROW AS othername ...`). Since PostgreSQL allows trigger procedures to be written in any number of user-defined languages, access to the data is handled in a language-specific way.
- PostgreSQL only has row-level triggers, no statement-level triggers.
- PostgreSQL only allows the execution of a stored procedure for the triggered action. SQL99 allows the execution of a number of other SQL commands, such as `CREATE TABLE` as triggered action. This limitation is not hard to work around by creating a stored procedure that executes these commands.

SQL99 specifies that multiple triggers should be fired in time-of-creation order. PostgreSQL uses name order, which was judged more convenient to work with.

See Also

CREATE FUNCTION, *ALTER TRIGGER*, *DROP TRIGGER*, *PostgreSQL Programmer's Guide*

CREATE TYPE

Name

`CREATE TYPE` — define a new data type

Synopsis

```
CREATE TYPE typename ( INPUT = input_function, OUTPUT = output_function
      , INTERNALLENGTH = { internallength | VARIABLE }
    [ , DEFAULT = default ]
    [ , ELEMENT = element ] [ , DELIMITER = delimiter ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
)

CREATE TYPE typename AS
    ( column_name data_type [, ... ] )
```

Inputs

typename

The name (optionally schema-qualified) of a type to be created.

internallength

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

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_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 to be used between values in arrays made of this type.

default

The default value for the data type. Usually this is omitted, so that the default is NULL.

alignment

Storage alignment requirement of the data type. If specified, must be `char`, `int2`, `int4`, or `double`; the default is `int4`.

storage

Storage technique for the data type. If specified, must be `plain`, `external`, `extended`, or `main`; the default is `plain`.

column_name

The name of a column of the composite type.

data_type

The name of an existing data type.

Outputs

`CREATE TYPE`

Message returned if the type is successfully created.

Description

`CREATE TYPE` allows the user to register a new data type with PostgreSQL for use in the current data base. The user who defines a type becomes its owner.

If a schema name is given then the type is created in the specified schema. Otherwise it is created in the current schema (the one at the front of the search path; see `CURRENT_SCHEMA()`). The type name must be distinct from the name of any existing type or domain in the same schema. (Because tables have associated data types, type names also must not conflict with table names in the same schema.)

Base Types

The first form of `CREATE TYPE` creates a new base type (scalar type). It 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. The input function may be declared as taking one argument of type `cstring`, or as taking three arguments of types `cstring`, `OID`, `int4`. (The first argument is the input text as a C string, the second argument is the element type in case this is an array type, and the third is the `typmod` of the destination column, if known.) It should return a value of the data type itself. The output function may be declared as taking one argument of the new data type, or as taking two arguments of which the second is type `OID`. (The second argument is again the array element type for array types.) The output function should return type `cstring`.

You should at this point be wondering how the input and output functions can be declared to have results or inputs of the new type, when they have to be created before the new type can be created. The answer is that the input function must be created first, then the output function, then the data type. PostgreSQL will first see the name of the new data type as the return type of the input function. It will create a "shell" type, which is simply a placeholder entry in `pg_type`, and link the input function definition to the shell type. Similarly the output function will be linked to the (now already existing) shell type. Finally, `CREATE TYPE` replaces the shell entry with a complete type definition, and the new type can be used.

> **Note:** In PostgreSQL versions before 7.3, it was customary to avoid creating a shell type by replacing the functions' forward references to the type name with the placeholder pseudo-type `OPAQUE`. The `cstring` inputs and results also had to be declared as `OPAQUE` before 7.3. To support loading of old dump files, `CREATE TYPE` will accept functions declared using `opaque`, but it will issue a NOTICE and change the function's declaration to use the correct types.

New base data types can be fixed length, in which case *`internallength`* is a positive integer, or variable length, indicated by setting *`internallength`* to `VARIABLE`. (Internally, this is represented by setting `typlen` to -1.) The internal representation of all variable-length types must start with an integer giving the total length of this value of the type.

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

```
ELEMENT = int4
```

More details about array types appear below.

To indicate the delimiter to be used between values in the external representation of arrays of this type, *`delimiter`* can be set to a specific character. The default delimiter is the comma (`','`). Note that the delimiter is associated with the array element type, not the array type itself.

A default value may be specified, in case a user wants columns of the data type to default to something other than NULL. Specify the default with the `DEFAULT` keyword. (Such a default may be overridden by an explicit `DEFAULT` clause attached to a particular column.)

The optional flag, `PASSEDBYVALUE`, indicates that values of this data type are passed by value rather than by reference. Note that you may not pass by value types whose internal representation is longer than the width of the `Datum` type (four bytes on most machines, eight bytes on a few).

The *`alignment`* keyword specifies the storage alignment required for the data type. The allowed values equate to alignment on 1, 2, 4, or 8 byte boundaries. Note that variable-length types must have an alignment of at least 4, since they necessarily contain an `int4` as their first component.

The *`storage`* keyword allows selection of storage strategies for variable-length data types (only `plain` is allowed for fixed-length types). `plain` disables TOAST for the data type: it will always be stored in-line and not compressed. `extended` gives full TOAST capability: the system will first try to compress a long data value, and will move the value out of the main table row if it's still too long. `external` allows the value to be moved out of the main table, but the system will not try to compress it. `main` allows compression, but discourages moving the value out of the main table. (Data items with this storage method may still be moved out of the main table if there is no other way to make a row fit, but they will be kept in the main table preferentially over `extended` and `external` items.)

Composite Types

The second form of `CREATE TYPE` creates a composite type. The composite type is specified by a list of column names and data types. This is essentially the same as the row type of a table, but using `CREATE TYPE` avoids the need to create an actual table when all that is wanted is to define a type. A stand-alone composite type is useful as the return type of a function.

Array Types

Whenever a user-defined base data type is created, PostgreSQL automatically creates an associated array type, whose name consists of the base type's name prepended with an underscore. The parser understands this naming convention, and translates requests for columns of type `foo[]` into requests for type `_foo`. The implicitly-created array type is variable length and uses the built-in input and output functions `array_in` and `array_out`.

You might reasonably ask "why is there an `ELEMENT` option, if the system makes the correct array type automatically?" The only case where it's useful to use `ELEMENT` is when you are making a fixed-length type that happens to be internally an array of N identical things, and you want to allow the N things to be accessed directly by subscripting, in addition to whatever operations you plan to provide for the type as a whole. For example, type `name` allows its constituent `chars` to be accessed this way. A 2-D `point` type could allow its two component floats to be accessed like `point[0]` and `point[1]`. Note that this facility only works for fixed-length types whose internal form is exactly a sequence of N identical fixed-length fields. A subscriptable variable-length type must have the generalized internal representation used by `array_in` and `array_out`. For historical reasons (i.e., this is clearly wrong but it's far too late to change it), subscripting of fixed-length array types starts from zero, rather than from one as for variable-length arrays.

Notes

User-defined type names cannot begin with the underscore character ("_") and can only be 62 characters long (or in general `NAMEDATALEN-2`, rather than the `NAMEDATALEN-1` characters allowed for other names). Type names beginning with underscore are reserved for internally-created array type names.

Examples

This example creates the `box` data type and then uses the type in a table definition:

```
CREATE TYPE box (INTERNALLENGTH = 16,
    INPUT = my_procedure_1, OUTPUT = my_procedure_2);
CREATE TABLE myboxes (id INT4, description box);
```

If `box`'s internal structure were an array of four `float4`s, we might instead say

```
CREATE TYPE box (INTERNALLENGTH = 16,
    INPUT = my_procedure_1, OUTPUT = my_procedure_2,
    ELEMENT = float4);
```

which would allow a box value's component floats to be accessed by subscripting. Otherwise the type behaves the same as before.

This example creates a large object type and uses it in a table definition:

```
CREATE TYPE bigobj (INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE);
CREATE TABLE big_objs (id int4, obj bigobj);
```

This example creates a composite type and uses it in a table function definition:

```
    CREATE TYPE compfoo AS (f1 int, f2 text);
    CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS ’SELECT fooid, fooname FROM foo’ L
GUAGE SQL;
```

Compatibility

This `CREATE TYPE` command is a PostgreSQL extension. There is a `CREATE TYPE` statement in SQL99 that is rather different in detail.

See Also

CREATE FUNCTION, *DROP TYPE*, *PostgreSQL Programmer’s Guide*

CREATE USER

Name

`CREATE USER` — define a new database user account

Synopsis

```
CREATE USER username [ [ WITH ] option [ ... ] ]

where option can be:

      SYSID uid
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | CREATEDB | NOCREATEDB
    | CREATEUSER | NOCREATEUSER
    | IN GROUP groupname [, ...]
    | VALID UNTIL 'abstime'
```

Description

`CREATE USER` will add a new user to an instance of PostgreSQL. Refer to the *Administrator's Guide* for information about managing users and authentication. You must be a database superuser to use this command.

Parameters

`username`

The name of the user.

`uid`

The `SYSID` clause can be used to choose the PostgreSQL user ID of the user that is being created. It is not at all necessary that those match the Unix user IDs, but some people choose to keep the numbers the same.

If this is not specified, the highest assigned user ID plus one (with a minimum of 100) will be used as default.

`password`

Sets the user's password. If you do not plan to use password authentication you can omit this option, but the user won't be able to connect to a password-authenticated server. The password can be set or changed later, using *ALTER USER*.

`ENCRYPTED`
`UNENCRYPTED`

These keywords control whether the password is stored encrypted in `pg_shadow`. (If neither is specified, the default behavior is determined by the `PASSWORD_ENCRYPTION` server parameter.) If the presented string is already in MD5-encrypted format, then it is stored as-is, regardless of whether `ENCRYPTED` or `UNENCRYPTED` is specified. This allows reloading of encrypted passwords during dump/restore.

See the chapter on client authentication in the *Administrator’s Guide* for details on how to set up authentication mechanisms. Note that older clients may lack support for the MD5 authentication mechanism that is needed to work with passwords that are stored encrypted.

`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 himself. This option will also make the user a superuser who can override all access restrictions. 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. Multiple group names may be listed.

`abstime`

The `VALID UNTIL` clause sets an absolute time after which the user’s password is no longer valid. If this clause is omitted the login will be valid for all time.

Diagnostics

`CREATE USER`

Message returned if the command completes successfully.

Notes

Use *ALTER USER* to change the attributes of a user, and *DROP USER* to remove a user. Use *ALTER GROUP* to add the user to groups or remove the user from groups. PostgreSQL includes a program *createuser* that has the same functionality as this command (in fact, it calls this command) but can be run from the command shell.

Examples

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

The `CREATE USER` statement is a PostgreSQL extension. The SQL standard leaves the definition of users to the implementation.

See Also

ALTER USER, *DROP USER*, createuser

CREATE VIEW

Name

`CREATE VIEW` — define a new view

Synopsis

```
    CREATE [ OR REPLACE ] VIEW view [ ( column name list ) ] AS SELECT query
```

Inputs

`view`

The name (optionally schema-qualified) of a view to be created.

`column name list`

An optional list of names to be used for columns of the view. If given, these names override the column names that would be deduced from the SQL query.

`query`

An SQL query (that is, a `SELECT` statement) which will provide the columns and rows of the view.

Refer to *SELECT* for more information about valid arguments.

Outputs

`CREATE VIEW`

The message returned if the view is successfully created.

`ERROR: Relation 'view' already exists`

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

`WARNING: Attribute '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 a warning:

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

whereas this command does not:

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

Description

`CREATE VIEW` defines a view of a query. The view is not physically materialized. Instead, a query rewrite rule (an `ON SELECT` rule) is automatically generated to support SELECT operations on views.

`CREATE OR REPLACE VIEW` is similar, but if a view of the same name already exists, it is replaced. You can only replace a view with a new query that generates the identical set of columns (i.e., same column names and data types).

If a schema name is given (for example, `CREATE VIEW myschema.myview ...`) then the view is created in the specified schema. Otherwise it is created in the current schema (the one at the front of the search path; see `CURRENT_SCHEMA()`). The view name must be distinct from the name of any other view, table, sequence, or index in the same schema.

Notes

Currently, views are read only: the system will not allow an insert, update, or delete on a view. You can get the effect of an updatable view by creating rules that rewrite inserts, etc. on the view into appropriate actions on other tables. For more information see *CREATE RULE*.

Use the `DROP VIEW` statement to drop views.

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
(2 rows)
```

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 ]
```

The optional clauses for the full SQL92 command are:

CHECK OPTION

: This option is to do with updatable views. All `INSERT` and `UPDATE` commands 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.

`CREATE OR REPLACE VIEW` is a PostgreSQL language extension.

DEALLOCATE

Name

`DEALLOCATE` — remove a prepared query

Synopsis

```
DEALLOCATE [ PREPARE ] plan_name
```

Inputs

PREPARE

This keyword is ignored.

plan_name

The name of the prepared query to remove.

Outputs

`DEALLOCATE`

The prepared query was removed successfully.

Description

`DEALLOCATE` is used to remove a previously prepared query. If you do not explicitly `DEALLOCATE` a prepared query, it is removed when the session ends.

For more information on prepared queries, see *PREPARE*.

Compatibility

SQL92

SQL92 includes a `DEALLOCATE` statement, but it is only for use in embedded SQL clients.

DECLARE

Name

`DECLARE` — define a cursor

Synopsis

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

Inputs

cursorname

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 PostgreSQL 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 PostgreSQL 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 read only mode. Since this is the only cursor access mode available in PostgreSQL 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 PostgreSQL this keyword provokes an informational error message.

column

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

Outputs

`DECLARE CURSOR`

The message returned if the SELECT is run successfully.

`WARNING: Closing pre-existing portal "`*`cursorname`*`"`

This message is reported if the same cursor name was already declared in the current transaction block. The previous definition is discarded.

`ERROR: DECLARE CURSOR 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 format using *FETCH*.

Normal cursors return data in text format, either ASCII or another encoding scheme depending on how the PostgreSQL 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 need to convert it to a binary format to manipulate it. BINARY cursors give you back the data in the native binary representation.

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`).

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.

String representation is architecture-neutral whereas binary representation can differ between different machine architectures. *PostgreSQL does not resolve byte ordering or representation issues for binary cursors*. 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. However, binary cursors may be a little more efficient since there is less conversion overhead in the server to client data transfer.

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. Use to *BEGIN*, *COMMIT* and *ROLLBACK* to define a transaction block.

In SQL92 cursors are only available in embedded SQL (ESQL) applications. The PostgreSQL backend does not implement an explicit `OPEN cursor` statement; a cursor is considered to be open when it is declared. However, ecpg, the embedded SQL preprocessor for PostgreSQL, supports the SQL92 cursor 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. PostgreSQL permits cursors to be used interactively. SQL92 allows embedded or modular cursors to update database information. All PostgreSQL cursors are read only. The BINARY keyword is a PostgreSQL extension.

DELETE

Name

`DELETE` — delete rows of a table

Synopsis

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

Inputs

`table`

The name (optionally schema-qualified) 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 clause from the specified table.

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

> **Tip:** *TRUNCATE* is a PostgreSQL extension which provides a faster mechanism to remove all rows from a table.

By default DELETE will delete tuples in the table specified and all its sub-tables. If you wish to only update the specific table mentioned, you should use the ONLY clause.

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 PostgreSQL are read-only.

DROP AGGREGATE

Name

`DROP AGGREGATE` — remove a user-defined aggregate function

Synopsis

```
DROP AGGREGATE name ( type ) [ CASCADE | RESTRICT ]
```

Inputs

`name`

The name (optionally schema-qualified) of an existing aggregate function.

`type`

The input data type of the aggregate function, or `*` if the function accepts any input type. (Refer to the *PostgreSQL User's Guide* for further information about data types.)

CASCADE

Automatically drop objects that depend on the aggregate.

RESTRICT

Refuse to drop the aggregate if there are any dependent objects. This is the default.

Outputs

`DROP AGGREGATE`

Message returned if the command is successful.

`ERROR: RemoveAggregate: aggregate 'name' for type type does not exist`

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

Description

`DROP AGGREGATE` will delete an existing aggregate definition. To execute this command the current user must be the owner of the aggregate.

Notes

Use *CREATE AGGREGATE* 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; the statement is a PostgreSQL language extension.

DROP CAST

Name

`DROP CAST` — remove a user-defined cast

Synopsis

```
DROP CAST (sourcetype AS targettype)
    [ CASCADE | RESTRICT ]
```

Description

`DROP CAST` removes a previously defined cast.

To be able to drop a cast, you must own the source or the target data type. These are the same privileges that are required to create a cast.

Parameters

`sourcetype`

The name of the source data type of the cast.

`targettype`

The name of the target data type of the cast.

`CASCADE`
`RESTRICT`

These key words do not have any effect, since there are no dependencies on casts.

Notes

Use `CREATE CAST` to create user-defined casts.

Examples

To drop the cast from type `text` to type `int`:

```
DROP CAST (text AS int4);
```

Compatibility

The `DROP CAST` command conforms to SQL99.

See Also

CREATE CAST

DROP CONVERSION

Name

`DROP CONVERSION` — remove a user-defined conversion

Synopsis

```
DROP CONVERSION conversion_name
    [ CASCADE | RESTRICT ]
```

Description

`DROP CONVERSION` removes a previously defined conversion.

To be able to drop a conversion, you must own the conversion.

Parameters

`conversion_name`

The name of the conversion. The conversion name may be schema-qualified.

`CASCADE`
`RESTRICT`

These key words do not have any effect, since there are no dependencies on conversions.

Notes

Use `CREATE CONVERSION` to create user-defined conversions.

The privileges required to drop a conversion may be changed in a future release.

Examples

To drop the conversion named `myname`:

```
DROP CONVERSION myname;
```

Compatibility

`DROP CONVERSION` is a PostgreSQL extension. There is no `DROP CONVERSION` statement in SQL99.

See Also

CREATE CONVERSION

DROP DATABASE

Name

`DROP DATABASE` — remove a database

Synopsis

```
DROP DATABASE name
```

Inputs

name

The name of an existing database to remove.

Outputs

`DROP DATABASE`

This message is returned if the command is successful.

`DROP DATABASE: cannot be executed on the currently open database`

You cannot be connected to the database you are about to remove. Instead, connect to `template1` or any other database and run this command again.

`DROP DATABASE: may not be called in a transaction block`

You must finish the transaction in progress before you can call this command.

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 owner (usually the user that created it).

`DROP DATABASE` cannot be undone. Use it with care!

Notes

This command cannot be executed while connected to the target database. Thus, it might be more convenient to use the shell script *dropdb*, which is a wrapper around this command, instead.

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

Compatibility

SQL92

`DROP DATABASE` statement is a PostgreSQL language extension; there is no such command in SQL92.

DROP DOMAIN

Name

`DROP DOMAIN` — remove a user-defined domain

Synopsis

```
DROP DOMAIN domainname [, ...]  [ CASCADE | RESTRICT ]
```

Inputs

`domainname`

The name (optionally schema-qualified) of an existing domain.

`CASCADE`

Automatically drop objects that depend on the domain (such as table columns).

`RESTRICT`

Refuse to drop the domain if there are any dependent objects. This is the default.

Outputs

`DROP DOMAIN`

The message returned if the command is successful.

`ERROR: RemoveDomain: type 'domainname' does not exist`

This message occurs if the specified domain (or type) is not found.

Description

`DROP DOMAIN` will remove a user domain from the system catalogs.

Only the owner of a domain can remove it.

Examples

To remove the `box` domain:

```
DROP DOMAIN box;
```

Compatibility

SQL92

See Also

CREATE DOMAIN

DROP FUNCTION

Name

`DROP FUNCTION` — remove a user-defined function

Synopsis

```
DROP FUNCTION name ( [ type [, ...] ] ) [ CASCADE | RESTRICT ]
```

Inputs

name

The name (optionally schema-qualified) of an existing function.

type

The type of a parameter of the function.

CASCADE

Automatically drop objects that depend on the function (such as operators or triggers).

RESTRICT

Refuse to drop the function if there are any dependent objects. This is the default.

Outputs

`DROP FUNCTION`

Message returned if the command completes successfully.

`WARNING: 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 the definition of an existing function. To execute this command the user must be the owner of the function. The input argument types to the function must be specified, since several different functions may exist with the same name and different argument lists.

Notes

Refer to *CREATE FUNCTION* for information on creating functions.

Examples

This command removes the square root function:

```
DROP FUNCTION sqrt(integer);
```

Compatibility

A `DROP FUNCTION` statement is defined in SQL99. One of its syntax forms is similar to PostgreSQL's.

See Also

CREATE FUNCTION

DROP GROUP

Name

`DROP GROUP` — remove a user group

Synopsis

```
DROP GROUP name
```

Inputs

name

The name of an existing group.

Outputs

`DROP GROUP`

The message returned if the group is successfully deleted.

Description

`DROP GROUP` removes the specified group from the database. The users in the group are not deleted.

Use *CREATE GROUP* to add new groups, and *ALTER GROUP* to change a group's membership.

Usage

To drop a group:

```
DROP GROUP staff;
```

Compatibility

SQL92

There is no `DROP GROUP` in SQL92.

DROP INDEX

Name

`DROP INDEX` — remove an index

Synopsis

```
DROP INDEX index_name [, ...] [ CASCADE | RESTRICT ]
```

Inputs

`index_name`

The name (optionally schema-qualified) of an index to remove.

CASCADE

Automatically drop objects that depend on the index.

RESTRICT

Refuse to drop the index if there are any dependent objects. This is the default.

Outputs

`DROP INDEX`

The message returned if the command completes successfully.

`ERROR: index "index_name" does not exist`

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 PostgreSQL language extension.

Refer to *CREATE INDEX* 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` — remove a user-defined procedural language

Synopsis

```
DROP [ PROCEDURAL ] LANGUAGE name [ CASCADE | RESTRICT ]
```

Inputs

name

The name of an existing procedural language. For backward compatibility, the name may be enclosed by single quotes.

CASCADE

Automatically drop objects that depend on the language (such as functions in the language).

RESTRICT

Refuse to drop the language if there are any dependent objects. This is the default.

Outputs

`DROP LANGUAGE`

This message is returned if the language is successfully dropped.

`ERROR: Language "name" doesn’t exist`

This message occurs if a language called *name* is not found in the database.

Description

`DROP PROCEDURAL LANGUAGE` will remove the definition of the previously registered procedural language called *name*.

Notes

The `DROP PROCEDURAL LANGUAGE` statement is a PostgreSQL language extension.

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

Usage

This command removes the PL/Sample language:

```
DROP LANGUAGE plsample;
```

Compatibility

SQL92

There is no `DROP PROCEDURAL LANGUAGE` in SQL92.

DROP OPERATOR

Name

`DROP OPERATOR` — remove a user-defined operator

Synopsis

```
    DROP OPERATOR id ( lefttype | NONE , righttype | NONE ) [ CASCADE | RE-
STRICT ]
```

Inputs

`id`

The identifier (optionally schema-qualified) of an existing operator.

`lefttype`

The type of the operator's left argument; write `NONE` if the operator has no left argument.

`righttype`

The type of the operator's right argument; write `NONE` if the operator has no right argument.

CASCADE

Automatically drop objects that depend on the operator.

RESTRICT

Refuse to drop the operator if there are any dependent objects. This is the default.

Outputs

`DROP OPERATOR`

The message returned if the command is successful.

`ERROR: RemoveOperator: binary operator 'oper' taking 'lefttype' and 'righttype' does not exist`

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

`ERROR: RemoveOperator: left unary operator 'oper' taking 'lefttype' does not exist`

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

`ERROR: RemoveOperator: right unary operator 'oper' taking 'righttype' does not exist`

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

Description

`DROP OPERATOR` 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, must be specified as `NONE`.

Notes

The `DROP OPERATOR` statement is a PostgreSQL language extension.

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

Usage

Remove power operator `a^n` for `int4`:

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

Remove left unary negation operator (`! b`) for `boolean`:

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

Remove right unary factorial operator (`i !`) for `int4`:

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

Compatibility

SQL92

There is no `DROP OPERATOR` in SQL92.

DROP OPERATOR CLASS

Name

`DROP OPERATOR CLASS` — remove a user-defined operator class

Synopsis

```
DROP OPERATOR CLASS name USING access_method [ CASCADE | RESTRICT ]
```

Inputs

`name`

The name (optionally schema-qualified) of an existing operator class.

`access_method`

The name of the index access method the operator class is for.

CASCADE

Automatically drop objects that depend on the operator class.

RESTRICT

Refuse to drop the operator class if there are any dependent objects. This is the default.

Outputs

`DROP OPERATOR CLASS`

The message returned if the command is successful.

Description

`DROP OPERATOR CLASS` drops an existing operator class from the database. To execute this command you must be the owner of the operator class.

Notes

The `DROP OPERATOR CLASS` statement is a PostgreSQL language extension.

Refer to *CREATE OPERATOR CLASS* for information on how to create operator classes.

Usage

Remove B-tree operator class `widget_ops`:

```
    DROP OPERATOR CLASS widget_ops USING btree;
```

This command will not execute if there are any existing indexes that use the operator class. Add `CASCADE` to drop such indexes along with the operator class.

Compatibility

SQL92

There is no `DROP OPERATOR CLASS` in SQL92.

DROP RULE

Name

`DROP RULE` — remove a rewrite rule

Synopsis

```
DROP RULE name ON relation [ CASCADE | RESTRICT ]
```

Inputs

name

The name of an existing rule to drop.

relation

The name (optionally schema-qualified) of the relation the rule applies to.

CASCADE

Automatically drop objects that depend on the rule.

RESTRICT

Refuse to drop the rule if there are any dependent objects. This is the default.

Outputs

`DROP RULE`

Message returned if successful.

`ERROR: Rule "name" not found`

This message occurs if the specified rule does not exist.

Description

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

Notes

The `DROP RULE` statement is a PostgreSQL language extension.

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

Usage

To drop the rewrite rule `newrule`:

```
DROP RULE newrule ON mytable;
```

Compatibility

SQL92

There is no `DROP RULE` in SQL92.

DROP SCHEMA

Name

`DROP SCHEMA` — remove a schema

Synopsis

```
    DROP SCHEMA name [, ...] [ CASCADE | RESTRICT ]
```

Inputs

`name`

The name of a schema.

CASCADE

Automatically drop objects (tables, functions, etc) that are contained in the schema.

RESTRICT

Refuse to drop the schema if it contains any objects. This is the default.

Outputs

`DROP SCHEMA`

The message returned if the schema is successfully dropped.

`ERROR: Schema "name" does not exist`

This message occurs if the specified schema does not exist.

Description

`DROP SCHEMA` removes schemas from the data base.

A schema can only be dropped by its owner or a superuser. Note that the owner can drop the schema (and thereby all contained objects) even if he does not own some of the objects within the schema.

Notes

Refer to the `CREATE SCHEMA` statement for information on how to create a schema.

Usage

To remove schema `mystuff` from the database, along with everything it contains:

```
    DROP SCHEMA mystuff CASCADE;
```

Compatibility

SQL92

`DROP SCHEMA` is fully compatible with SQL92, except that the standard only allows one schema to be dropped per command.

DROP SEQUENCE

Name

`DROP SEQUENCE` — remove a sequence

Synopsis

```
DROP SEQUENCE name [, ...] [ CASCADE | RESTRICT ]
```

Inputs

`name`

The name (optionally schema-qualified) of a sequence.

CASCADE

Automatically drop objects that depend on the sequence.

RESTRICT

Refuse to drop the sequence if there are any dependent objects. This is the default.

Outputs

`DROP SEQUENCE`

The message returned if the sequence is successfully dropped.

`ERROR: sequence "name" 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 PostgreSQL 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` — remove a table

Synopsis

```
DROP TABLE name [, ...] [ CASCADE | RESTRICT ]
```

Inputs

name

The name (optionally schema-qualified) of an existing table to drop.

CASCADE

Automatically drop objects that depend on the table (such as views).

RESTRICT

Refuse to drop the table if there are any dependent objects. This is the default.

Outputs

`DROP TABLE`

The message returned if the command completes successfully.

`ERROR: table "name" does not exist`

If the specified table does not exist in the database.

Description

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

`DROP TABLE` always removes any indexes, rules, triggers, and constraints that exist for the target table. However, to drop a table that is referenced by a foreign-key constraint of another table, CASCADE must be specified. (CASCADE will remove the foreign-key constraint, not the other table itself.)

Notes

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

Usage

To destroy two tables, `films` and `distributors`:

```
DROP TABLE films, distributors;
```

Compatibility

SQL92

DROP TRIGGER

Name

`DROP TRIGGER` — remove a trigger

Synopsis

```
DROP TRIGGER name ON table [ CASCADE | RESTRICT ]
```

Inputs

name

The name of an existing trigger.

table

The name (optionally schema-qualified) of a table.

CASCADE

Automatically drop objects that depend on the trigger.

RESTRICT

Refuse to drop the trigger if there are any dependent objects. This is the default.

Outputs

`DROP TRIGGER`

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 an existing trigger definition. To execute this command the current user must be the owner of the table for which the trigger is defined.

Examples

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.

SQL99

The `DROP TRIGGER` statement in PostgreSQL is incompatible with SQL99. In SQL99, trigger names are not local to tables, so the command is simply `DROP TRIGGER` *`name`*.

See Also

CREATE TRIGGER

DROP TYPE

Name

`DROP TYPE` — remove a user-defined data type

Synopsis

```
DROP TYPE typename [, ...] [ CASCADE | RESTRICT ]
```

Inputs

`typename`

The name (optionally schema-qualified) of an existing type.

CASCADE

Automatically drop objects that depend on the type (such as table columns, functions, operators, etc).

RESTRICT

Refuse to drop the type if there are any dependent objects. This is the default.

Outputs

`DROP TYPE`

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.

Examples

To remove the `box` type:

```
DROP TYPE box;
```

Compatibility

Note that the `CREATE TYPE` command and the data type extension mechanisms in PostgreSQL differ from SQL99.

See Also

CREATE TYPE

DROP USER

Name

`DROP USER` — remove a database user account

Synopsis

```
DROP USER name
```

Description

`DROP USER` removes the specified user from the database. It does not remove tables, views, or other objects owned by the user. If the user owns any database, an error is raised.

Parameters

name

The name of an existing user.

Diagnostics

`DROP USER`

The message returned if the user is successfully deleted.

`ERROR: DROP USER: user "name" does not exist`

This message occurs if the user name is not found.

`DROP USER: user "name" owns database "name", cannot be removed`

You must drop the database first or change its ownership.

Notes

Use *CREATE USER* to add new users, and *ALTER USER* to change a user's attributes. PostgreSQL includes a program *dropuser* that has the same functionality as this command (in fact, it calls this command) but can be run from the command shell.

Examples

To drop a user account:

```
DROP USER jonathan;
```

Compatibility

The `DROP USER` statement is a PostgreSQL extension. The SQL standard leaves the definition of users to the implementation.

See Also

CREATE USER, *ALTER USER*, dropuser

DROP VIEW

Name

`DROP VIEW` — remove a view

Synopsis

```
DROP VIEW name [, ...] [ CASCADE | RESTRICT ]
```

Inputs

`name`

The name (optionally schema-qualified) of an existing view.

CASCADE

Automatically drop objects that depend on the view (such as other views).

RESTRICT

Refuse to drop the view if there are any dependent objects. This is the default.

Outputs

`DROP VIEW`

The message returned if the command is successful.

`ERROR: view name does not exist`

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

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

END

Name

`END` — commit the current transaction

Synopsis

```
END [ WORK | TRANSACTION ]
```

Inputs

WORK
TRANSACTION

Optional keywords. They have no effect.

Outputs

`COMMIT`

Message returned if the transaction is successfully committed.

`WARNING: COMMIT: no transaction in progress`

If there is no transaction in progress.

Description

`END` is a PostgreSQL extension, and is a synonym for the SQL92-compatible *COMMIT*.

Notes

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

Use *ROLLBACK* to abort a transaction.

Usage

To make all changes permanent:

```
END WORK;
```

Compatibility

SQL92

`END` is a PostgreSQL extension which provides functionality equivalent to *COMMIT*.

EXECUTE

Name

`EXECUTE` — execute a prepared query

Synopsis

```
EXECUTE plan_name [ (parameter [, ...] ) ]
```

Inputs

`plan_name`

The name of the prepared query to execute.

`parameter`

The actual value of a parameter to the prepared query. This must be an expression yielding a value of a type compatible with the data-type specified for this parameter position in the `PREPARE` statement that created the prepared query.

Description

`EXECUTE` is used to execute a previously prepared query. Since prepared queries only exist for the duration of a session, the prepared query must have been created by a `PREPARE` statement executed earlier in the current session.

If the `PREPARE` statement that created the query specified some parameters, a compatible set of parameters must be passed to the `EXECUTE` statement, or else an error is raised. Note that (unlike functions) prepared queries are not overloaded based on the type or number of their parameters: the name of a prepared query must be unique within a database session.

For more information on the creation and usage of prepared queries, see *PREPARE*.

Compatibility

SQL92

SQL92 includes an `EXECUTE` statement, but it is only for use in embedded SQL clients. The `EXECUTE` statement implemented by PostgreSQL also uses a somewhat different syntax.

EXPLAIN

Name

`EXPLAIN` — show the execution plan of a statement

Synopsis

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] query
```

Inputs

ANALYZE

Flag to carry out the query and show actual run times.

VERBOSE

Flag to show detailed query plan dump.

`query`

Any *`query`*.

Outputs

Query plan

Explicit query plan from the PostgreSQL planner.

Note: Prior to PostgreSQL 7.3, the query plan was emitted in the form of a NOTICE message. Now it appears as a query result (formatted like a table with a single text column).

Description

This command displays the execution plan that the PostgreSQL planner generates for the supplied query. The execution plan shows how the table(s) referenced by the query will be scanned---by plain sequential scan, index scan, etc.---and if multiple tables are referenced, what join algorithms will be used to bring together the required tuples from each input table.

The most critical part of the display is the estimated query execution cost, which is the planner's guess at how long it will take to run the query (measured in units of disk page fetches). Actually two numbers are shown: the start-up time before the first tuple can be returned, and the total time to return all the tuples. For most queries the total time is what matters, but in contexts such as an EXISTS subquery the planner will choose the smallest start-up time instead of the smallest total time (since the executor will stop after getting one tuple, anyway). Also, if you limit the number of tuples to return with a LIMIT clause, the planner makes an appropriate interpolation between the endpoint costs to estimate which plan is really the cheapest.

The ANALYZE option causes the query to be actually executed, not only planned. The total elapsed time expended within each plan node (in milliseconds) and total number of rows it actually returned are added to the display. This is useful for seeing whether the planner's estimates are close to reality.

Caution

Keep in mind that the query is actually executed when ANALYZE is used. Although `EXPLAIN` will discard any output that a SELECT would return, other side-effects of the query will happen as usual. If you wish to use `EXPLAIN ANALYZE` on an INSERT, UPDATE, or DELETE query without letting the query affect your data, use this approach:

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

The VERBOSE option emits the full internal representation of the plan tree, rather than just a summary. Usually this option is only useful for debugging PostgreSQL. The VERBOSE dump is either pretty-printed or not, depending on the setting of the `EXPLAIN_PRETTY_PRINT` configuration parameter.

Notes

There is only sparse documentation on the optimizer's use of cost information in PostgreSQL. Refer to the *User's Guide* and *Programmer's Guide* for more information.

Usage

To show a query plan for a simple query on a table with a single `int4` column and 10000 rows:

```
    EXPLAIN SELECT * FROM foo;
                                QUERY PLAN
---------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

If there is an index and we use a query with an indexable WHERE condition, `EXPLAIN` will show a different plan:

```
    EXPLAIN SELECT * FROM foo WHERE i = 4;
                                  QUERY PLAN
--------------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
```

```
   Index Cond: (i = 4)
(2 rows)
```

And here is an example of a query plan for a query using an aggregate function:

```
    EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;
                                     QUERY PLAN
---------------------------------------------------------------------
 Aggregate  (cost=23.93..23.93 rows=1 width=4)
   ->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6 width=4)
         Index Cond: (i < 10)
(3 rows)
```

Note that the specific numbers shown, and even the selected query strategy, may vary between PostgreSQL releases due to planner improvements.

Compatibility

SQL92

There is no `EXPLAIN` statement defined in SQL92.

FETCH

Name

`FETCH` — retrieve rows from a table using a cursor

Synopsis

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

Inputs

`direction`

`selector` defines the fetch direction. It can be one of 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 specifies 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:

`WARNING: PerformPortalFetch: portal "*cursor*" not found`

If *`cursor`* is not previously declared. The cursor must be declared within a transaction block.

`WARNING: FETCH/ABSOLUTE not supported, using RELATIVE`

PostgreSQL does not support absolute positioning of cursors.

`ERROR: FETCH/RELATIVE at current position is not supported`

SQL92 allows one to repetitively retrieve the cursor at its "current position" using the syntax

```
FETCH RELATIVE 0 FROM cursor.
```

PostgreSQL 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, PostgreSQL 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 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`.

Notes

Note that the FORWARD and BACKWARD keywords are PostgreSQL extensions. The SQL92 syntax is also supported, specified in the second form of the command. See below for details on compatibility issues.

Updating data in a cursor is not supported by PostgreSQL, 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.

Use *MOVE* to change cursor position. *DECLARE* will define a cursor. Refer to *BEGIN*, *COMMIT*, and *ROLLBACK* for further information about transactions.

Usage

The following examples traverses a table using a cursor.

```
    -- 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

SQL92

Note: The non-embedded use of cursors is a PostgreSQL extension. The syntax and usage of cursors is being compared against the embedded form of cursors defined in 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 PostgreSQL are relative numbers so this capability is not supported.

:*variable*

  Target host variable(s).

GRANT

Name

`GRANT` — define access privileges

Synopsis

```
    GRANT { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIG-
GER }
        [,...] | ALL [ PRIVILEGES ] }
        ON [ TABLE ] tablename [, ...]
        TO { username | GROUP groupname | PUBLIC } [, ...]

    GRANT { { CREATE | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
        ON DATABASE dbname [, ...]
        TO { username | GROUP groupname | PUBLIC } [, ...]

    GRANT { EXECUTE | ALL [ PRIVILEGES ] }
        ON FUNCTION funcname ([type, ...]) [, ...]
        TO { username | GROUP groupname | PUBLIC } [, ...]

    GRANT { USAGE | ALL [ PRIVILEGES ] }
        ON LANGUAGE langname [, ...]
        TO { username | GROUP groupname | PUBLIC } [, ...]

    GRANT { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
        ON SCHEMA schemaname [, ...]
        TO { username | GROUP groupname | PUBLIC } [, ...]
```

Description

The `GRANT` command gives specific permissions on an object (table, view, sequence, database, function, procedural language, or schema) to one or more users or groups of users. These permissions are added to those already granted, if any.

The key word `PUBLIC` indicates that the privileges are to be granted to all users, including those that may be created later. `PUBLIC` may be thought of as an implicitly defined group that always includes all users. Note that any particular user will have the sum of privileges granted directly to him, privileges granted to any group he is presently a member of, and privileges granted to `PUBLIC`.

There is no need to grant privileges to the creator of an object, as the creator has all privileges by default. (The creator could, however, choose to revoke some of his own privileges for safety.) Note that the ability to grant and revoke privileges is inherent in the creator and cannot be lost. The right to drop an object, or to alter it in any way not described by a grantable right, is likewise inherent in the creator, and cannot be granted or revoked.

Depending on the type of object, the initial default privileges may include granting some privileges to `PUBLIC`. The default is no public access for tables and schemas; `TEMP` table creation privilege for databases; `EXECUTE` privilege for functions; and `USAGE` privilege for languages. The object creator may of course revoke these privileges. (For maximum security, issue the `REVOKE` in the same transaction that creates the object; then there is no window in which another user may use the object.)

The possible privileges are:

SELECT

Allows *SELECT* from any column of the specified table, view, or sequence. Also allows the use of *COPY* TO. For sequences, this privilege also allows the use of the `currval` function.

INSERT

Allows *INSERT* of a new row into the specified table. Also allows *COPY* FROM.

UPDATE

Allows *UPDATE* of any column of the specified table. `SELECT ... FOR UPDATE` also requires this privilege (besides the `SELECT` privilege). For sequences, this privilege allows the use of the `nextval` and `setval` functions.

DELETE

Allows *DELETE* of a row from the specified table.

RULE

Allows the creation of a rule on the table/view. (See *CREATE RULE* statement.)

REFERENCES

To create a foreign key constraint, it is necessary to have this privilege on both the referencing and referenced tables.

TRIGGER

Allows the creation of a trigger on the specified table. (See *CREATE TRIGGER* statement.)

CREATE

For databases, allows new schemas to be created within the database.

For schemas, allows new objects to be created within the schema. To rename an existing object, you must own the object *and* have this privilege for the containing schema.

TEMPORARY
TEMP

Allows temporary tables to be created while using the database.

EXECUTE

Allows the use of the specified function and the use of any operators that are implemented on top of the function. This is the only type of privilege that is applicable to functions. (This syntax works for aggregate functions, as well.)

USAGE

For procedural languages, allows the use of the specified language for the creation of functions in that language. This is the only type of privilege that is applicable to procedural languages.

For schemas, allows access to objects contained in the specified schema (assuming that the objects' own privilege requirements are also met). Essentially this allows the grantee to "look up" objects within the schema.

ALL PRIVILEGES

Grant all of the privileges applicable to the object at once. The `PRIVILEGES` key word is optional in PostgreSQL, though it is required by strict SQL.

The privileges required by other commands are listed on the reference page of the respective command.

Notes

The *REVOKE* command is used to revoke access privileges.

It should be noted that database *superusers* can access all objects regardless of object privilege settings. This is comparable to the rights of `root` in a Unix system. As with `root`, it's unwise to operate as a superuser except when absolutely necessary.

Currently, to grant privileges in PostgreSQL to only a few columns, you must create a view having the desired columns and then grant privileges to that view.

Use psql's `\dp` command to obtain information about existing privileges, for example:

```
lusitania=> \dp mytable
        Access privileges for database "lusitania"
 Schema |  Table  |           Access privileges
--------+---------+---------------------------------------
 public | mytable | {=r,miriam=arwdRxt,"group todos=arw"}
(1 row)
```

The entries shown by `\dp` are interpreted thus:

```
              =xxxx -- privileges granted to PUBLIC
         uname=xxxx -- privileges granted to a user
   group gname=xxxx -- privileges granted to a group

                  r -- SELECT ("read")
                  w -- UPDATE ("write")
                  a -- INSERT ("append")
                  d -- DELETE
                  R -- RULE
                  x -- REFERENCES
                  t -- TRIGGER
                  X -- EXECUTE
                  U -- USAGE
                  C -- CREATE
                  T -- TEMPORARY
            arwdRxt -- ALL PRIVILEGES (for tables)
```

The above example display would be seen by user `miriam` after creating table `mytable` and doing

```
GRANT SELECT ON mytable TO PUBLIC;
GRANT SELECT,UPDATE,INSERT ON mytable TO GROUP todos;
```

If the "Access privileges" column is empty for a given object, it means the object has default privileges (that is, its privileges field is NULL). Default privileges always include all privileges for the owner, and may include some privileges for `PUBLIC` depending on the object type, as explained above. The first `GRANT` or `REVOKE` on an object will instantiate the default privileges (producing, for example, `{=,miriam=arwdRxt}`) and then modify them per the specified request.

Examples

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 PRIVILEGES ON kinds TO manuel;
```

Compatibility

SQL92

The `PRIVILEGES` key word in `ALL PRIVILEGES` is required. SQL does not support setting the privileges on more than one table per command.

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 ]
```

SQL allows to grant the USAGE privilege on other kinds of objects: CHARACTER SET, COLLATION, TRANSLATION, DOMAIN.

The TRIGGER privilege was introduced in SQL99. The RULE privilege is a PostgreSQL extension.

See Also

REVOKE

INSERT

Name

`INSERT` — create new rows in a table

Synopsis

```
    INSERT INTO table [ ( column [, ...] ) ]
        { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) | SE-
LECT query }
```

Inputs

`table`

The name (optionally schema-qualified) of an existing table.

`column`

The name of a column in *`table`*.

DEFAULT VALUES

All columns will be filled by null values or by values specified when the table was created using `DEFAULT` clauses.

`expression`

A valid expression or value to assign to *`column`*.

`DEFAULT`

This column will be filled in by the column DEFAULT clause, or NULL if a default is not available.

`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 numeric OID of the inserted row.

`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 a time or several rows as a result of a query. The columns in the target list may be listed in any order.

Each column not present in the target list will be inserted using a default value, either a declared DEFAULT value or NULL. PostgreSQL will reject the new column if a NULL is inserted into a column declared NOT 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`:

```
INSERT INTO films VALUES
    (’UA502’,’Bananas’,105,’1971-07-13’,’Comedy’,INTERVAL ’82 minute’);
```

In this second example the last column `len` is omitted and therefore it will have the default value of NULL:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES (’T_601’, ’Yojimbo’, 106, DATE ’1961-06-16’, ’Drama’);
```

In the third example, we use the DEFAULT values for the date columns rather than specifying an entry.

```
INSERT INTO films VALUES
    (’UA502’,’Bananas’,105,DEFAULT,’Comedy’,INTERVAL ’82 minute’);
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES (’T_601’, ’Yojimbo’, 106, DEFAULT, ’Drama’);
```

Insert a single row into table distributors; note that only column `name` is specified, so the omitted 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 (refer to the *PostgreSQL User's Guide* for further information about arrays):

```
-- Create an empty 3x3 gameboard for noughts-and-crosses
-- (all of these queries create the same board attribute)
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

`INSERT` is fully compatible with SQL92. Possible limitations in features of the *`query`* clause are documented for *SELECT*.

LISTEN

Name

`LISTEN` — listen for a notification

Synopsis

```
LISTEN name
```

Inputs

name

Name of notify condition.

Outputs

`LISTEN`

Message returned upon successful completion of registration.

`WARNING: Async_Listen: We are already listening on name`

If this backend is already registered for that notify condition.

Description

`LISTEN` registers the current PostgreSQL backend as a listener on the notify condition *name*.

Whenever the command `NOTIFY name` 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 unregistered 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 PostgreSQL application programming interface it uses. With the 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.

NOTIFY contains a more extensive discussion of the use of `LISTEN` and `NOTIFY`.

Notes

`name` 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 63 characters long.

In some previous releases of PostgreSQL, *`name`* 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:

```
LISTEN virtual;
NOTIFY virtual;

Asynchronous NOTIFY ’virtual’ from backend with pid ’8448’ received.
```

Compatibility

SQL92

There is no `LISTEN` in SQL92.

LOAD

Name

`LOAD` — load or reload a shared library file

Synopsis

```
    LOAD 'filename'
```

Description

Loads a shared library file into the PostgreSQL backend's address space. If the file had been loaded previously, it is first unloaded. This command is primarily useful to unload and reload a shared library file that has been changed since the backend first loaded it. To make use of the shared library, function(s) in it need to be declared using the *CREATE FUNCTION* command.

The file name is specified in the same way as for shared library names in *CREATE FUNCTION*; in particular, one may rely on a search path and automatic addition of the system's standard shared library file name extension. See the *Programmer's Guide* for more detail.

Compatibility

`LOAD` is a PostgreSQL extension.

See Also

CREATE FUNCTION, *PostgreSQL Programmer's Guide*

LOCK

Name

`LOCK` — explicitly lock a table

Synopsis

```
LOCK [ TABLE ] name [, ...]
LOCK [ TABLE ] name [, ...] IN lockmode MODE

where lockmode is one of:

 ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE |
 SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

Inputs

name

The name (optionally schema-qualified) of an existing table to lock.

ACCESS SHARE MODE

This is the least restrictive lock mode. It conflicts only with ACCESS EXCLUSIVE mode. It is used to protect a table from being modified by concurrent `ALTER TABLE`, `DROP TABLE` and `VACUUM FULL` commands.

> **Note:** The `SELECT` command acquires a lock of this mode on referenced tables. In general, any query that only reads a table and does not modify it will acquire this lock mode.

ROW SHARE MODE

Conflicts with EXCLUSIVE and ACCESS EXCLUSIVE lock modes.

> **Note:** The `SELECT FOR UPDATE` command acquires a lock of this mode on the target table(s) (in addition to `ACCESS SHARE` locks on any other tables that are referenced but not selected `FOR UPDATE`).

ROW EXCLUSIVE MODE

Conflicts with SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes.

> **Note:** The commands `UPDATE`, `DELETE`, and `INSERT` acquire this lock mode on the target table (in addition to `ACCESS SHARE` locks on any other referenced tables). In general, this lock mode will be acquired by any query that modifies the data in a table.

SHARE UPDATE EXCLUSIVE MODE

Conflicts with SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes. This mode protects a table against concurrent schema changes and `VACUUM` runs.

> **Note:** Acquired by `VACUUM` (without `FULL`).

SHARE MODE

Conflicts with ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes. This mode protects a table against concurrent data changes.

> **Note:** Acquired by `CREATE INDEX`.

SHARE ROW EXCLUSIVE MODE

Conflicts with ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes.

> **Note:** This lock mode is not automatically acquired by any PostgreSQL command.

EXCLUSIVE MODE

Conflicts with ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes. This mode allows only concurrent ACCESS SHARE, i.e., only reads from the table can proceed in parallel with a transaction holding this lock mode.

> **Note:** This lock mode is not automatically acquired by any PostgreSQL command.

ACCESS EXCLUSIVE MODE

Conflicts with all lock modes. This mode guarantees that the holder is the only transaction accessing the table in any way.

> **Note:** Acquired by `ALTER TABLE`, `DROP TABLE`, and `VACUUM FULL` statements. This is also the default lock mode for `LOCK TABLE` statements that do not specify a mode explicitly.

Outputs

`LOCK TABLE`

The lock was successfully acquired.

`ERROR name: Table does not exist.`

Message returned if *`name`* does not exist.

Description

`LOCK TABLE` obtains a table-level lock, waiting if necessary for any conflicting locks to be released. Once obtained, the lock is held for the remainder of the current transaction. (There is no `UNLOCK TABLE` command; locks are always released at transaction end.)

When acquiring locks automatically for commands that reference tables, PostgreSQL always uses the least restrictive lock mode possible. `LOCK TABLE` provides for cases when you might need more restrictive locking.

For example, suppose an application runs a transaction at READ COMMITTED isolation level and needs to ensure that data in a table remains stable for the duration of the transaction. To achieve this you could obtain SHARE lock mode over the table before querying. This will prevent concurrent data changes and ensure subsequent reads of the table see a stable view of committed data, because SHARE lock mode conflicts with the ROW EXCLUSIVE lock acquired by writers, and your `LOCK TABLE` *`name`* `IN SHARE MODE` statement will wait until any concurrent holders of ROW EXCLUSIVE mode commit or roll back. Thus, once you obtain the lock, there are no uncommitted writes outstanding; furthermore none can begin until you release the lock.

> **Note:** To achieve a similar effect when running a transaction at the SERIALIZABLE isolation level, you have to execute the `LOCK TABLE` statement before executing any DML statement. A serializable transaction's view of data will be frozen when its first DML statement begins. A later `LOCK` will still prevent concurrent writes --- but it won't ensure that what the transaction reads corresponds to the latest committed values.

If a transaction of this sort is going to change the data in the table, then it should use SHARE ROW EXCLUSIVE lock mode instead of SHARE mode. This ensures that only one transaction of this type runs at a time. Without this, a deadlock is possible: two transactions might both acquire SHARE mode, and then be unable to also acquire ROW EXCLUSIVE mode to actually perform their updates. (Note that a transaction's own locks never conflict, so a transaction can acquire ROW EXCLUSIVE mode when it holds SHARE mode --- but not if anyone else holds SHARE mode.)

Two general rules may be followed to prevent deadlock conditions:

- Transactions have to acquire locks on the same objects in the same order.

  For example, if one application updates row R1 and than updates row R2 (in the same transaction) then the second application shouldn't update row R2 if it's going to update row R1 later (in a single transaction). Instead, it should update rows R1 and R2 in the same order as the first application.

- If multiple lock modes are involved for a single object, then transactions should always acquire the most restrictive mode first.

  An example for this rule was given previously when discussing the use of SHARE ROW EXCLUSIVE mode rather than SHARE mode.

PostgreSQL does detect deadlocks and will rollback at least one waiting transaction to resolve the deadlock. If it is not practical to code an application to follow the above rules strictly, an alternative solution is to be prepared to retry transactions when they are aborted by deadlocks.

When locking multiple tables, the command `LOCK a, b;` is equivalent to `LOCK a; LOCK b;`. The tables are locked one-by-one in the order specified in the `LOCK` command.

Notes

`LOCK ... IN ACCESS SHARE MODE` requires `SELECT` privileges on the target table. All other forms of `LOCK` require `UPDATE` and/or `DELETE` privileges.

`LOCK` is useful only inside a transaction block (`BEGIN`...`COMMIT`), since the lock is dropped as soon as the transaction ends. A `LOCK` command appearing outside any transaction block forms a self-contained transaction, so the lock will be dropped as soon as it is obtained.

RDBMS locking uses the following standard terminology:

EXCLUSIVE

An exclusive lock prevents other locks of the same type from being granted.

SHARE

A shared lock allows others to also hold the same type of lock, but prevents the corresponding EXCLUSIVE lock from being granted.

ACCESS

Locks table schema.

ROW

Locks individual rows.

PostgreSQL does not follow this terminology exactly. `LOCK TABLE` only deals with table-level locks, and so the mode names involving ROW are all misnomers. These mode names should generally be read as indicating the intention of the user to acquire row-level locks within the locked table. Also, ROW EXCLUSIVE mode does not follow this naming convention accurately, since it is a sharable table lock. Keep in mind that all the lock modes have identical semantics so far as `LOCK TABLE` is concerned, differing only in the rules about which modes conflict with which.

Usage

Obtain a SHARE lock on a primary key table when going to perform inserts into a foreign key table:

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = ’Star Wars: Episode I - The Phantom Menace’;
-- Do ROLLBACK if record was not returned
INSERT INTO films_user_comments VALUES
    (_id_, ’GREAT! I was waiting for it for so long!’);
COMMIT WORK;
```

Take a SHARE ROW EXCLUSIVE lock on a primary key table when going to perform a delete operation:

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

Compatibility

SQL92

There is no `LOCK TABLE` in SQL92, which instead uses `SET TRANSACTION` to specify concurrency levels on transactions. We support that too; see *SET TRANSACTION* for details.

Except for ACCESS SHARE, ACCESS EXCLUSIVE, and SHARE UPDATE EXCLUSIVE lock modes, the PostgreSQL lock modes and the `LOCK TABLE` syntax are compatible with those present in Oracle(TM).

MOVE

Name

`MOVE` — position a cursor on a specified row of a table

Synopsis

```
MOVE [ direction ] [ count ]
    { IN | 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 *FETCH* for details on syntax and usage.

Notes

`MOVE` is a PostgreSQL language extension.

Refer to *FETCH* for a description of valid arguments. Refer to *DECLARE* to define a cursor. Refer to *BEGIN*, *COMMIT*, and *ROLLBACK* 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 position.

NOTIFY

Name

`NOTIFY` — generate a notification

Synopsis

```
NOTIFY name
```

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 PostgreSQL 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 PostgreSQL 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. PostgreSQL keeps self-notifies separate from notifies arriving from other backends, so you cannot miss an outside notify by ignoring your own notifies.)

Notes

`name` can be any string valid as a name; it need not correspond to the name of any actual table. If *`name`* is enclosed in double-quotes, it need not even be a syntactically valid name, but can be any string up to 63 characters long.

In some previous releases of PostgreSQL, *`name`* 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 PostgreSQL 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:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous NOTIFY 'virtual' from backend with pid '8448' received.
```

Compatibility

SQL92

There is no `NOTIFY` statement in SQL92.

PREPARE

Name

`PREPARE` — create a prepared query

Synopsis

```
PREPARE plan_name [ (datatype [, ...] ) ] AS query
```

Inputs

plan_name

An arbitrary name given to this particular prepared query. It must be unique within a single session, and is used to execute or remove a previously prepared query.

datatype

The data-type of a parameter to the prepared query. To refer to the parameters in the prepared query itself, use `$1`, `$2`, etc.

Outputs

`PREPARE`

The query has been prepared successfully.

Description

`PREPARE` creates a prepared query. A prepared query is a server-side object that can be used to optimize performance. When the `PREPARE` statement is executed, the specified query is parsed, rewritten, and planned. When a subsequent `EXECUTE` statement is issued, the prepared query need only be executed. Thus, the parsing, rewriting, and planning stages are only performed once, instead of every time the query is executed.

Prepared queries can take parameters: values that are substituted into the query when it is executed. To specify the parameters to a prepared query, include a list of data-types with the `PREPARE` statement. In the query itself, you can refer to the parameters by position using `$1`, `$2`, etc. When executing the query, specify the actual values for these parameters in the `EXECUTE` statement -- refer to *EXECUTE* for more information.

Prepared queries are stored locally (in the current backend), and only exist for the duration of the current database session. When the client exits, the prepared query is forgotten, and so it must be re-created before being used again. This also means that a single prepared query cannot be used by multiple simultaneous database clients; however, each client can create their own prepared query to use.

Prepared queries have the largest performance advantage when a single backend is being used to execute a large number of similar queries. The performance difference will be particularly significant if the queries are complex to plan or rewrite. For example, if the query involves a join of many tables or requires the application of several rules. If the query is relatively simple to plan and rewrite but relatively expensive to execute, the performance advantage of prepared queries will be less noticeable.

Notes

In some situations, the query plan produced by PostgreSQL for a prepared query may be inferior to the plan produced if the query were submitted and executed normally. This is because when the query is planned (and the optimizer attempts to determine the optimal query plan), the actual values of any parameters specified in the query are unavailable. PostgreSQL collects statistics on the distribution of data in the table, and can use constant values in a query to make guesses about the likely result of executing the query. Since this data is unavailable when planning prepared queries with parameters, the chosen plan may be sub-optimal.

For more information on query planning and the statistics collected by PostgreSQL for query optimization purposes, see the *ANALYZE* documentation.

Compatibility

SQL92

SQL92 includes a `PREPARE` statement, but it is only for use in embedded SQL clients. The `PREPARE` statement implemented by PostgreSQL also uses a somewhat different syntax.

REINDEX

Name

`REINDEX` — rebuild corrupted indexes

Synopsis

```
REINDEX { TABLE | DATABASE | INDEX } name [ FORCE ]
```

Inputs

TABLE

Recreate all indexes of a specified table.

DATABASE

Recreate all system indexes of a specified database. (User-table indexes are not included.)

INDEX

Recreate a specified index.

`name`

The name of the specific table/database/index to be reindexed. Table and index names may be schema-qualified.

FORCE

Force rebuild of system indexes. Without this keyword `REINDEX` skips system indexes that are not marked invalid. FORCE is irrelevant for `REINDEX INDEX`, or when reindexing user indexes.

Outputs

`REINDEX`

Message returned if the table is successfully reindexed.

Description

`REINDEX` is used to rebuild corrupted indexes. Although in theory this should never be necessary, in practice indexes may become corrupted due to software bugs or hardware failures. `REINDEX` provides a recovery method.

`REINDEX` also removes certain dead index pages that can't be reclaimed any other way. See the "Routine Reindexing" section in the manual for more information.

If you suspect corruption of an index on a user table, you can simply rebuild that index, or all indexes on the table, using `REINDEX INDEX` or `REINDEX TABLE`.

> **Note:** Another approach to dealing with a corrupted user-table index is just to drop and recreate it. This may in fact be preferable if you would like to maintain some semblance of normal operation on the table meanwhile. `REINDEX` acquires exclusive lock on the table, while `CREATE INDEX` only locks out writes not reads of the table.

Things are more difficult if you need to recover from corruption of an index on a system table. In this case it's important for the backend doing the recovery to not have used any of the suspect indexes itself. (Indeed, in this sort of scenario you may find that backends are crashing immediately at start-up, due to reliance on the corrupted indexes.) To recover safely, the postmaster must be shut down and a stand-alone PostgreSQL backend must be started instead, giving it the command-line options -O and -P (these options allow system table modifications and prevent use of system indexes, respectively). Then issue `REINDEX INDEX`, `REINDEX TABLE`, or `REINDEX DATABASE` depending on how much you want to reconstruct. If in doubt, use `REINDEX DATABASE FORCE` to force reconstruction of all system indexes in the database. Then quit the standalone backend and restart the postmaster.

Since this is likely the only situation when most people will ever use a standalone backend, some usage notes might be in order:

- Start the backend with a command like

  ```
  postgres -D $PGDATA -O -P my_database
  ```

  Provide the correct path to the database area with `-D`, or make sure that the environment variable `PGDATA` is set. Also specify the name of the particular database you want to work in.
- You can issue any SQL command, not only `REINDEX`.
- Be aware that the standalone backend treats newline as the command entry terminator; there is no intelligence about semicolons, as there is in psql. To continue a command across multiple lines, you must type backslash just before each newline except the last one. Also, you won't have any of the conveniences of command-line editing (no command history, for example).
- To quit the backend, type EOF (**Control**+**D**, usually).

See the postgres reference page for more information.

Usage

Recreate the indexes on the table `mytable`:

```
REINDEX TABLE mytable;
```

Rebuild a single index:

```
REINDEX INDEX my_index;
```

Rebuild all system indexes (this will only work in a standalone backend):

```
REINDEX DATABASE my_database FORCE;
```

Compatibility

SQL92

There is no `REINDEX` in SQL92.

RESET

Name

`RESET` — restore the value of a run-time parameter to a default value

Synopsis

```
RESET variable
```

```
RESET ALL
```

Inputs

variable

The name of a run-time parameter. See *SET* for a list.

ALL

Resets all settable run-time parameters to default values.

Description

`RESET` restores run-time parameters to their default values. Refer to *SET* for details. `RESET` is an alternate spelling for

```
SET variable TO DEFAULT
```

The default value is defined as the value that the variable would have had, had no `SET` ever been issued for it in the current session. The actual source of this value might be a compiled-in default, the postmaster's configuration file or command-line switches, or per-database or per-user default settings. See the *Administrator's Guide* for details.

See the `SET` manual page for details on the transaction behavior of `RESET`.

Diagnostics

See under the *SET* command.

Examples

Set `DateStyle` to its default value:

```
RESET DateStyle;
```

Set `geqo` to its default value:

```
RESET GEQO;
```

Compatibility

`RESET` is a PostgreSQL extension.

REVOKE

Name

`REVOKE` — remove access privileges

Synopsis

```
    REVOKE { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIG-
GER }
        [,...] | ALL [ PRIVILEGES ] }
        ON [ TABLE ] tablename [, ...]
        FROM { username | GROUP groupname | PUBLIC } [, ...]

    REVOKE { { CREATE | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
        ON DATABASE dbname [, ...]
        FROM { username | GROUP groupname | PUBLIC } [, ...]

    REVOKE { EXECUTE | ALL [ PRIVILEGES ] }
        ON FUNCTION funcname ([type, ...]) [, ...]
        FROM { username | GROUP groupname | PUBLIC } [, ...]

    REVOKE { USAGE | ALL [ PRIVILEGES ] }
        ON LANGUAGE langname [, ...]
        FROM { username | GROUP groupname | PUBLIC } [, ...]

    REVOKE { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
        ON SCHEMA schemaname [, ...]
        FROM { username | GROUP groupname | PUBLIC } [, ...]
```

Description

`REVOKE` allows the creator of an object to revoke previously granted permissions from one or more users or groups of users. The key word `PUBLIC` refers to the implicitly defined group of all users.

Note that any particular user will have the sum of privileges granted directly to him, privileges granted to any group he is presently a member of, and privileges granted to `PUBLIC`. Thus, for example, revoking SELECT privilege from `PUBLIC` does not necessarily mean that all users have lost SELECT privilege on the object: those who have it granted directly or via a group will still have it.

See the description of the *GRANT* command for the meaning of the privilege types.

Notes

Use psql's `\z` command to display the privileges granted on existing objects. See also *GRANT* for information about the format.

Examples

Revoke insert privilege for the public on table `films`:

```
    REVOKE INSERT ON films FROM PUBLIC;
```

Revoke all privileges from user `manuel` on view `kinds`:

```
    REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

Compatibility

SQL92

The compatibility notes of the *GRANT* command apply analogously to `REVOKE`. The syntax summary is:

```
    REVOKE [ GRANT OPTION FOR ] { SELECT | INSERT | UPDATE | DELETE | REFER-
ENCES }
        ON object [ ( column [, ...] ) ]
        FROM { PUBLIC | username [, ...] }
        { RESTRICT | CASCADE }
```

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 tries to revoke this privilege it fails if he specifies the RESTRICT keyword.

See Also

GRANT

ROLLBACK

Name

`ROLLBACK` — abort the current transaction

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ]
```

Inputs

None.

Outputs

`ROLLBACK`

Message returned if successful.

`WARNING: ROLLBACK: no transaction in progress`

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

Use *COMMIT* to successfully terminate a transaction. *ABORT* is a synonym for `ROLLBACK`.

Usage

To abort all changes:

```
ROLLBACK WORK;
```

Compatibility

SQL92

SQL92 only specifies the two forms `ROLLBACK` and `ROLLBACK WORK`. Otherwise full compatibility.

SELECT

Name

`SELECT` — retrieve rows from a table or view

Synopsis

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
    [ FOR UPDATE [ OF tablename [, ...] ] ]

where from_item can be:

[ ONLY ] table_name [ * ]
    [ [ AS ] alias [ ( column_alias_list ) ] ]
|
( select )
    [ AS ] alias [ ( column_alias_list ) ]
|
table_function_name ( [ argument [, ...] ] )
    [ AS ] alias [ ( column_alias_list | column_definition_list ) ]
|
table_function_name ( [ argument [, ...] ] )
    AS ( column_definition_list )
|
from_item [ NATURAL ] join_type from_item
    [ ON join_condition | USING ( join_column_list ) ]
```

Inputs

expression

The name of a table's column or an expression.

output_name

Specifies another name for an output column using the AS clause. This name is primarily used to label the column for display. It can also be used to refer to the column's value in ORDER BY and GROUP BY clauses. But the *output_name* cannot be used in the WHERE or HAVING clauses; write out the expression instead.

from_item

A table reference, sub-SELECT, table function, or JOIN clause. See below for details.

`condition`

A Boolean expression giving a result of true or false. See the WHERE and HAVING clause descriptions below.

`select`

A select statement with all features except the ORDER BY, LIMIT/OFFSET, and FOR UPDATE clauses (even those can be used when the select is parenthesized).

FROM items can contain:

`table_name`

The name (optionally schema-qualified) of an existing table or view. If `ONLY` is specified, only that table is scanned. If `ONLY` is not specified, the table and all its descendant tables (if any) are scanned. `*` can be appended to the table name to indicate that descendant tables are to be scanned, but in the current version, this is the default behavior. (In releases before 7.1, `ONLY` was the default behavior.) The default behavior can be modified by changing the `SQL_INHERITANCE` configuration option.

`alias`

A substitute name for the FROM item containing the alias. An alias is used for brevity or to eliminate ambiguity for self-joins (where the same table is scanned multiple times). When an alias is provided, it completely hides the actual name of the table or table function; for example given `FROM foo AS f`, the remainder of the SELECT must refer to this FROM item as `f` not `foo`. If an alias is written, a column alias list can also be written to provide substitute names for one or more columns of the table.

`select`

A sub-SELECT can appear in the FROM clause. This acts as though its output were created as a temporary table for the duration of this single SELECT command. Note that the sub-SELECT must be surrounded by parentheses, and an alias *must* be provided for it.

`table function`

A table function can appear in the FROM clause. This acts as though its output were created as a temporary table for the duration of this single SELECT command. An alias may also be used. If an alias is written, a column alias list can also be written to provide substitute names for one or more columns of the table function. If the table function has been defined as returning the `record` data type, an alias, or the keyword `AS`, must be present, followed by a column definition list in the form (`column_name` `data_type` [, ...]). The column definition list must match the actual number and types of columns returned by the function.

`join_type`

One of `[ INNER ] JOIN`, `LEFT [ OUTER ] JOIN`, `RIGHT [ OUTER ] JOIN`, `FULL [ OUTER ] JOIN`, or `CROSS JOIN`. For INNER and OUTER join types, exactly one of NATURAL, ON `join_condition`, or USING (`join_column_list`) must appear. For CROSS JOIN, none of these items may appear.

`join_condition`

A qualification condition. This is similar to the WHERE condition except that it only applies to the two from_items being joined in this JOIN clause.

`join_column_list`

A USING column list (a, b, ...) is shorthand for the ON condition left_table.a = right_table.a AND left_table.b = right_table.b ...

Outputs

Rows

The complete set of rows resulting from the query specification.

`count`

The count of rows returned by the query.

Description

`SELECT` will return rows from one or more tables. Candidates for selection are rows which satisfy the WHERE condition; if WHERE is omitted, all rows are candidates. (See *WHERE Clause*.)

Actually, the returned rows are not directly the rows produced by the FROM/WHERE/GROUP BY/HAVING clauses; rather, the output rows are formed by computing the SELECT output expressions for each selected row. `*` can be written in the output list as a shorthand for all the columns of the selected rows. Also, one can write *`table_name`*`.*` as a shorthand for the columns coming from just that table.

`DISTINCT` will eliminate duplicate rows from the result. `ALL` (the default) will return all candidate rows, including duplicates.

`DISTINCT ON` eliminates rows that match on all the specified expressions, keeping only the first row of each set of duplicates. The DISTINCT ON expressions are interpreted using the same rules as for ORDER BY items; see below. Note that the “first row” of each set is unpredictable unless `ORDER BY` is used to ensure that the desired row appears first. For example,

```
SELECT DISTINCT ON (location) location, time, report
FROM weatherReports
ORDER BY location, time DESC;
```

retrieves the most recent weather report for each location. But if we had not used ORDER BY to force descending order of time values for each location, we’d have gotten a report of unpredictable age for each location.

The GROUP BY clause allows a user to divide a table into groups of rows that match on one or more values. (See *GROUP BY Clause*.)

The HAVING clause allows selection of only those groups of rows meeting the specified condition. (See *HAVING Clause*.)

The ORDER BY clause causes the returned rows to be sorted in a specified order. If ORDER BY is not given, the rows are returned in whatever order the system finds cheapest to produce. (See *ORDER BY Clause*.)

SELECT queries can be combined using UNION, INTERSECT, and EXCEPT operators. Use parentheses if necessary to determine the ordering of these operators.

The UNION operator computes the collection of rows returned by the queries involved. Duplicate rows are eliminated unless ALL is specified. (See *UNION Clause*.)

The INTERSECT operator computes the rows that are common to both queries. Duplicate rows are eliminated unless ALL is specified. (See *INTERSECT Clause*.)

The EXCEPT operator computes the rows returned by the first query but not the second query. Duplicate rows are eliminated unless ALL is specified. (See *EXCEPT Clause*.)

The LIMIT clause allows a subset of the rows produced by the query to be returned to the user. (See *LIMIT Clause*.)

The FOR UPDATE clause causes the SELECT statement to lock the selected rows against concurrent updates.

You must have SELECT privilege to a table to read its values (See the `GRANT`/`REVOKE` statements). Use of FOR UPDATE requires UPDATE privilege as well.

FROM Clause

The FROM clause specifies one or more source tables for the SELECT. If multiple sources are specified, the result is conceptually the Cartesian product of all the rows in all the sources --- but usually qualification conditions are added to restrict the returned rows to a small subset of the Cartesian product.

When a FROM item is a simple table name, it implicitly includes rows from sub-tables (inheritance children) of the table. `ONLY` will suppress rows from sub-tables of the table. Before PostgreSQL 7.1, this was the default result, and adding sub-tables was done by appending `*` to the table name. This old behavior is available via the command `SET SQL_Inheritance TO OFF`.

A FROM item can also be a parenthesized sub-SELECT (note that an alias clause is required for a sub-SELECT!). This is an extremely handy feature since it's the only way to get multiple levels of grouping, aggregation, or sorting in a single query.

A FROM item can be a table function (typically, a function that returns multiple rows and/or columns, though actually any function can be used). The function is invoked with the given argument value(s), and then its output is scanned as though it were a table.

In some cases it is useful to define table functions that can return different column sets depending on how they are invoked. To support this, the table function can be declared as returning the pseudo-type `record`. When such a function is used in FROM, it must be followed by an alias, or the keyword `AS` alone, and then by a parenthesized list of column names and types. This provides a query-time composite type definition. The composite type definition must match the actual composite type returned from the function, or an error will be reported at run-time.

Finally, a FROM item can be a JOIN clause, which combines two simpler FROM items. (Use parentheses if necessary to determine the order of nesting.)

A CROSS JOIN or INNER JOIN is a simple Cartesian product, the same as you get from listing the two items at the top level of FROM. CROSS JOIN is equivalent to INNER JOIN ON (TRUE), that is, no rows are removed by qualification. These join types are just a notational convenience, since they do nothing you couldn't do with plain FROM and WHERE.

LEFT OUTER JOIN returns all rows in the qualified Cartesian product (i.e., all combined rows that pass its ON condition), plus one copy of each row in the left-hand table for which there was no right-hand row that passed the ON condition. This left-hand row is extended to the full width of the joined table by inserting null values for the right-hand columns. Note that only the `JOIN`'s own ON or USING condition is considered while deciding which rows have matches. Outer ON or WHERE conditions are applied afterwards.

Conversely, RIGHT OUTER JOIN returns all the joined rows, plus one row for each unmatched right-hand row (extended with nulls on the left). This is just a notational convenience, since you could convert it to a LEFT OUTER JOIN by switching the left and right inputs.

FULL OUTER JOIN returns all the joined rows, plus one row for each unmatched left-hand row (extended with nulls on the right), plus one row for each unmatched right-hand row (extended with nulls on the left).

For all the JOIN types except CROSS JOIN, you must write exactly one of ON *join_condition*, USING (*join_column_list*), or NATURAL. ON is the most general case: you can write any qualification expression involving the two tables to be joined. A USING column list (a, b, ...) is shorthand for the ON condition left_table.a = right_table.a AND left_table.b = right_table.b ... Also, USING implies that only one of each pair of equivalent columns will be included in the JOIN output, not both. NATURAL is shorthand for a USING list that mentions all similarly-named columns in the tables.

WHERE Clause

The optional WHERE condition has the general form:

```
WHERE boolean_expr
```

boolean_expr can consist of any expression which evaluates to a Boolean value. In many cases, this expression will be:

```
expr cond_op expr
```

or

```
log_op expr
```

where *cond_op* can be one of: =, <, <=, >, >= or <>, a conditional operator like ALL, ANY, IN, LIKE, or a locally defined operator, and *log_op* can be one of: AND, OR, NOT. SELECT will ignore all rows for which the WHERE condition does not return TRUE.

GROUP BY Clause

GROUP BY specifies a grouped table derived by the application of this clause:

```
GROUP BY expression [, ...]
```

GROUP BY will condense into a single row all selected rows that share the same values for the grouped columns. Aggregate functions, if any, are computed across all rows making up each group, producing a separate value for each group (whereas without GROUP BY, an aggregate produces a single value computed across all the selected rows). When GROUP BY is present, it is not valid for the SELECT output expression(s) to refer to ungrouped columns except within aggregate functions, since there would be more than one possible value to return for an ungrouped column.

A GROUP BY item can be an input column name, or the name or ordinal number of an output column (SELECT expression), or it can be an arbitrary expression formed from input-column values. In case of ambiguity, a GROUP BY name will be interpreted as an input-column name rather than an output column name.

HAVING Clause

The optional HAVING condition has the general form:

```
HAVING boolean_expr
```

where *boolean_expr* is the same as specified for the WHERE clause.

HAVING specifies a grouped table derived by the elimination of group rows that do not satisfy the *boolean_expr*. HAVING is different from WHERE: WHERE filters individual rows before application of GROUP BY, while HAVING filters group rows created by GROUP BY.

Each column referenced in *boolean_expr* shall unambiguously reference a grouping column, unless the reference appears within an aggregate function.

ORDER BY Clause

```
ORDER BY expression [ ASC | DESC | USING operator ] [, ...]
```

An ORDER BY item can be the name or ordinal number of an output column (SELECT expression), or it can be an arbitrary expression formed from input-column values. In case of ambiguity, an ORDER BY name will be interpreted as an output-column name.

The ordinal number refers to the ordinal (left-to-right) position of the result column. This feature makes it possible to define an ordering on the basis of a column that does not have a unique name. This is never absolutely necessary because it is always possible to assign a name to a result column using the AS clause, e.g.:

```
SELECT title, date_prod + 1 AS newlen FROM films ORDER BY newlen;
```

It is also possible to ORDER BY arbitrary expressions (an extension to SQL92), including fields that do not appear in the SELECT result list. Thus the following statement is legal:

```
SELECT name FROM distributors ORDER BY code;
```

A limitation of this feature is that an ORDER BY clause applying to the result of a UNION, INTERSECT, or EXCEPT query may only specify an output column name or number, not an expression.

Note that if an ORDER BY item is a simple name that matches both a result column name and an input column name, ORDER BY will interpret it as the result column name. This is the opposite of the choice that GROUP BY will make in the same situation. This inconsistency is mandated by the SQL92 standard.

Optionally one may add the key word `DESC` (descending) or `ASC` (ascending) after each column name in the `ORDER BY` clause. If not specified, `ASC` is assumed by default. Alternatively, a specific ordering operator name may be specified. `ASC` is equivalent to `USING <` and `DESC` is equivalent to `USING >`.

The null value sorts higher than any other value in a domain. In other words, with ascending sort order nulls sort at the end and with descending sort order nulls sort at the beginning.

Data of character types is sorted according to the locale-specific collation order that was established when the database cluster was initialized.

UNION Clause

```
table_query UNION [ ALL ] table_query
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
```

where *`table_query`* specifies any select expression without an ORDER BY, LIMIT, or FOR UPDATE clause. (ORDER BY and LIMIT can be attached to a sub-expression if it is enclosed in parentheses. Without parentheses, these clauses will be taken to apply to the result of the UNION, not to its right-hand input expression.)

The UNION operator computes the collection (set union) of the rows returned by the queries involved. The two SELECT statements that represent the direct operands of the UNION must produce the same number of columns, and corresponding columns must be of compatible data types.

The result of UNION does not contain any duplicate rows unless the ALL option is specified. ALL prevents elimination of duplicates.

Multiple UNION operators in the same SELECT statement are evaluated left to right, unless otherwise indicated by parentheses.

Currently, FOR UPDATE may not be specified either for a UNION result or for the inputs of a UNION.

INTERSECT Clause

```
table_query INTERSECT [ ALL ] table_query
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
```

where *`table_query`* specifies any select expression without an ORDER BY, LIMIT, or FOR UPDATE clause.

INTERSECT is similar to UNION, except that it produces only rows that appear in both query outputs, rather than rows that appear in either.

The result of INTERSECT does not contain any duplicate rows unless the ALL option is specified. With ALL, a row that has m duplicates in L and n duplicates in R will appear min(m,n) times.

Multiple INTERSECT operators in the same SELECT statement are evaluated left to right, unless parentheses dictate otherwise. INTERSECT binds more tightly than UNION --- that is, A UNION B INTERSECT C will be read as A UNION (B INTERSECT C) unless otherwise specified by parentheses.

EXCEPT Clause

```
table_query EXCEPT [ ALL ] table_query
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
```

where *table_query* specifies any select expression without an ORDER BY, LIMIT, or FOR UPDATE clause.

EXCEPT is similar to UNION, except that it produces only rows that appear in the left query's output but not in the right query's output.

The result of EXCEPT does not contain any duplicate rows unless the ALL option is specified. With ALL, a row that has m duplicates in L and n duplicates in R will appear max(m-n,0) times.

Multiple EXCEPT operators in the same SELECT statement are evaluated left to right, unless parentheses dictate otherwise. EXCEPT binds at the same level as UNION.

LIMIT Clause

```
    LIMIT { count | ALL }
    OFFSET start
```

where *count* specifies the maximum number of rows to return, and *start* specifies the number of rows to skip before starting to return rows.

LIMIT allows you to retrieve just a portion of the rows that are generated by the rest of the query. If a limit count is given, no more than that many rows will be returned. If an offset is given, that many rows will be skipped before starting to return rows.

When using LIMIT, it is a good idea to use an ORDER BY clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows---you may be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? You don't know what ordering unless you specify ORDER BY.

As of PostgreSQL 7.0, the query optimizer takes LIMIT into account when generating a query plan, so you are very likely to get different plans (yielding different row orders) depending on what you use for LIMIT and OFFSET. Thus, using different LIMIT/OFFSET values to select different subsets of a query result *will give inconsistent results* unless you enforce a predictable result ordering with ORDER BY. This is not a bug; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless ORDER BY is used to constrain the order.

FOR UPDATE Clause

```
        FOR UPDATE [ OF tablename [, ...] ]
```

FOR UPDATE causes the rows retrieved by the query to be locked as though for update. This prevents them from being modified or deleted by other transactions until the current transaction ends; that is, other transactions that attempt UPDATE, DELETE, or SELECT FOR UPDATE of these rows will be blocked until the current transaction ends. Also, if an UPDATE, DELETE, or SELECT FOR UPDATE from another transaction has already locked a selected row or rows, SELECT FOR UPDATE will wait for the other transaction to complete, and will then lock and return the updated row (or no row, if the row was deleted). For further discussion see the concurrency chapter of the *User's Guide*.

If specific tables are named in FOR UPDATE, then only rows coming from those tables are locked; any other tables used in the SELECT are simply read as usual.

FOR UPDATE cannot be used in contexts where returned rows can't be clearly identified with individual table rows; for example it can't be used with aggregation.

FOR UPDATE may appear before LIMIT for compatibility with pre-7.3 applications. However, it effectively executes after LIMIT, and so that is the recommended place to write it.

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
    (17 rows)
```

To sum the column `len` of all films and group the results 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
(5 rows)
```

To sum the column `len` of all films, group the results 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
(2 rows)
```

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
(13 rows)
```

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 wanted, 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
```

This example shows how to use a table function, both with and without a column definition list.

```
distributors:
 did |     name
-----+--------------
 108 | Westward
 111 | Walt Disney
 112 | Warner Bros.
 ...

CREATE FUNCTION distributors(int)
  RETURNS SETOF distributors AS '
  SELECT * FROM distributors WHERE did = $1;
  ' LANGUAGE SQL;

SELECT * FROM distributors(111);
 did |    name
-----+-------------
 111 | Walt Disney
(1 row)

CREATE FUNCTION distributors_2(int)
  RETURNS SETOF RECORD AS '
  SELECT * FROM distributors WHERE did = $1;
  ' LANGUAGE SQL;

SELECT * FROM distributors_2(111) AS (f1 int, f2 text);
 f1  |     f2
-----+-------------
 111 | Walt Disney
(1 row)
```

Compatibility

Extensions

PostgreSQL allows one to omit the `FROM` clause from a query. This feature was retained from the original PostQUEL query language. It has a straightforward use to compute the results of simple expressions:

```
SELECT 2+2;

 ?column?
----------
        4
```

Some other SQL databases cannot do this except by introducing a dummy one-row table to do the select from. A less obvious use is to abbreviate a normal select from one or more tables:

```
SELECT distributors.* WHERE distributors.name = ’Westward’;

 did | name
-----+----------
 108 | Westward
```

This works because an implicit FROM item is added for each table that is referenced in the query but not mentioned in FROM. While this is a convenient shorthand, it’s easy to misuse. For example, the query

```
SELECT distributors.* FROM distributors d;
```

is probably a mistake; most likely the user meant

```
SELECT d.* FROM distributors d;
```

rather than the unconstrained join

```
SELECT distributors.* FROM distributors d, distributors distributors;
```

that he will actually get. To help detect this sort of mistake, PostgreSQL 7.1 and later will warn if the implicit-FROM feature is used in a query that also contains an explicit FROM clause.

The table-function feature is a PostgreSQL extension.

SQL92

SELECT Clause

In the SQL92 standard, the optional keyword `AS` is just noise and can be omitted without affecting the meaning. The PostgreSQL parser requires this keyword when renaming output columns because the type extensibility features lead to parsing ambiguities in this context. `AS` is optional in FROM items, however.

The DISTINCT ON phrase is not part of SQL92. Nor are LIMIT and OFFSET.

In SQL92, an ORDER BY clause may only use result column names or numbers, while a GROUP BY clause may only use input column names. PostgreSQL extends each of these clauses to allow the other choice as well (but it uses the standard's interpretation if there is ambiguity). PostgreSQL also allows both clauses to specify arbitrary expressions. Note that names appearing in an expression will always be taken as input-column names, not as result-column names.

UNION/INTERSECT/EXCEPT Clause

The SQL92 syntax for UNION/INTERSECT/EXCEPT allows an additional CORRESPONDING BY option:

```
table_query UNION [ALL]
    [CORRESPONDING [BY (column [,...])]]
    table_query
```

The CORRESPONDING BY clause is not supported by PostgreSQL.

SELECT INTO

Name

`SELECT INTO` — create a new table from the results of a query

Synopsis

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS output_name ] [, ...]
    INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
    [ FOR UPDATE [ OF tablename [, ...] ] ]
```

Inputs

TEMPORARY
TEMP

If specified, the table is created as a temporary table. Refer to *CREATE TABLE* for details.

`new_table`

The name (optionally schema-qualified) of the table to be created.

All other inputs are described in detail for *SELECT*.

Outputs

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

Description

`SELECT INTO` creates a new table and fills it with data computed by a query. The data is not returned to the client, as it is with a normal `SELECT`. The new table's columns have the names and data types associated with the output columns of the `SELECT`.

Note: *CREATE TABLE AS* is functionally equivalent to `SELECT INTO`. `CREATE TABLE AS` is the recommended syntax, since `SELECT INTO` is not standard. In fact, this form of `SELECT INTO` is not available in PL/pgSQL or ecpg, because they interpret the INTO clause differently.

Compatibility

SQL92 uses `SELECT ... INTO` to represent selecting values into scalar variables of a host program, rather than creating a new table. This indeed is the usage found in PL/pgSQL and ecpg. The PostgreSQL usage of `SELECT INTO` to represent table creation is historical. It's best to use `CREATE TABLE AS` for this purpose in new code. (`CREATE TABLE AS` isn't standard either, but it's less likely to cause confusion.)

SET

Name

`SET` — change a run-time parameter

Synopsis

```
SET [ SESSION | LOCAL ] variable { TO | = } { value | 'value' | DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { timezone | LOCAL | DEFAULT }
```

Inputs

`SESSION`

Specifies that the command takes effect for the current session. (This is the default if neither `SESSION` nor `LOCAL` appears.)

`LOCAL`

Specifies that the command takes effect for only the current transaction. After `COMMIT` or `ROLLBACK`, the session-level setting takes effect again. Note that `SET LOCAL` will appear to have no effect if it's executed outside a `BEGIN` block, since the transaction will end immediately.

`variable`

A settable run-time parameter.

`value`

New value of parameter. `DEFAULT` can be used to specify resetting the parameter to its default value. Lists of strings are allowed, but more complex constructs may need to be single or double quoted.

Description

The `SET` command changes run-time configuration parameters. Many of the run-time parameters listed in the *Administrator's Guide* can be changed on-the-fly with `SET`. (But some require superuser privileges to change, and others cannot be changed after server or session start.) Note that `SET` only affects the value used by the current session.

If `SET` or `SET SESSION` is issued within a transaction that is later aborted, the effects of the `SET` command disappear when the transaction is rolled back. (This behavior represents a change from PostgreSQL versions prior to 7.3, where the effects of `SET` would not roll back after a later error.) Once the surrounding transaction is committed, the effects will persist until the end of the session, unless overridden by another `SET`.

The effects of `SET LOCAL` last only till the end of the current transaction, whether committed or not. A special case is `SET` followed by `SET LOCAL` within a single transaction: the `SET LOCAL` value will be seen until the end of the transaction, but afterwards (if the transaction is committed) the `SET` value will take effect.

Even with `autocommit` set to `off`, `SET` does not start a new transaction block. See the `autocommit` section of the *Administrator's Guide* for details.

Here are additional details about a few of the parameters that can be set:

`DATESTYLE`

Choose the date/time representation style. Two separate settings are involved: the default date/time output and the interpretation of ambiguous input.

The following are date/time output styles:

`ISO`

Use ISO 8601-style dates and times (`YYYY-MM-DD HH:MM:SS`). This is the default.

`SQL`

Use Oracle/Ingres-style dates and times. Note that this style has nothing to do with SQL (which mandates ISO 8601 style), the naming of this option is a historical accident.

`PostgreSQL`

Use traditional PostgreSQL format.

`German`

Use `dd.mm.yyyy` for numeric date representations.

The following two options determine both a substyle of the "SQL" and "PostgreSQL" output formats and the preferred interpretation of ambiguous date input.

`European`

Use `dd/mm/yyyy` for numeric date representations.

`NonEuropean`
`US`

Use `mm/dd/yyyy` for numeric date representations.

A value for `SET DATESTYLE` can be one from the first list (output styles), or one from the second list (substyles), or one from each separated by a comma.

`SET DATESTYLE` affects interpretation of input and provides several standard output formats. For applications needing different variations or tighter control over input or output, consider using the `to_char` family of functions.

There are several now-deprecated means for setting the date style in addition to the normal methods of setting it via `SET` or a configuration-file entry:

Setting the postmaster's `PGDATESTYLE` environment variable. (This will be overridden by any of the other methods.)
Running postmaster using the option `-o -e` to set dates to the `European` convention. (This overrides environment variables and configuration-file entries.)

Setting the client's `PGDATESTYLE` environment variable. If `PGDATESTYLE` is set in the frontend environment of a client based on libpq, libpq will automatically set `DATESTYLE` to the value of `PGDATESTYLE` during connection start-up. This is equivalent to a manually issued `SET DATESTYLE`.

NAMES

`SET NAMES` is an alias for `SET CLIENT_ENCODING`.

SEED

Sets the internal seed for the random number generator.

`value`

The value for the seed to be used by the `random` function. Allowed values are floating-point numbers between 0 and 1, which are then multiplied by 2^{31}-1.

The seed can also be set by invoking the `setseed` SQL function:

```
SELECT setseed(value);
```

SERVER_ENCODING

Shows the server-side multibyte encoding. (At present, this parameter can be shown but not set, because the encoding is determined at initdb time.)

TIME ZONE
TIMEZONE

Sets the default time zone for your session. Arguments can be an SQL time interval constant, an integer or double precision constant, or a string representing a time zone name recognized by the host operating system.

Here are some typical values for time zone settings:

'PST8PDT'

Set the time zone for Berkeley, California.

'Portugal'

Set the time zone for Portugal.

'Europe/Rome'

Set the time zone for Italy.

7

Set the time zone to 7 hours offset west from GMT (equivalent to PDT).

INTERVAL '08:00' HOUR TO MINUTE

Set the time zone to 8 hours offset west from GMT (equivalent to PST).

LOCAL
DEFAULT

Set the time zone to your local time zone (the one that your operating system defaults to).

The available time zone names depend on your operating system. For example, on Linux `/usr/share/zoneinfo` contains the database of time zones; the names of the files in that directory can be used as parameters to this command.

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

If the `PGTZ` environment variable is set in the frontend environment of a client based on libpq, libpq will automatically `SET TIMEZONE` to the value of `PGTZ` during connection start-up.

Use *SHOW* to show the current setting of a parameter.

Diagnostics

`SET`

Message returned if successful.

`ERROR: '`*`name`*` is not a valid option name`

The parameter you tried to set does not exist.

`ERROR: '`*`name`*`': permission denied`

You must be a superuser to alter certain settings.

`ERROR: '`*`name`*`' cannot be changed after server start`

Some parameters are fixed once the server is started.

Examples

Set the style of date to traditional PostgreSQL with European conventions:

```
SET DATESTYLE TO PostgreSQL,European;
```

Set the time zone for Berkeley, California, using quotes to preserve the uppercase spelling of the time zone name (note that the date style is `PostgreSQL` for this example):

```
SET TIME ZONE 'PST8PDT';
SELECT CURRENT_TIMESTAMP AS today;
               today
------------------------------------
 Tue Feb 26 07:32:21.42834 2002 PST
```

Set the time zone for Italy (note the required single quotes to handle the special characters):

```
SET TIME ZONE ’Europe/Rome’;
SELECT CURRENT_TIMESTAMP AS today;

             today
-------------------------------
 2002-10-08 05:39:35.008271+02
```

Compatibility

SQL92

`SET TIME ZONE` extends syntax defined in SQL9x. SQL9x allows only numeric time zone offsets while PostgreSQL allows full time zone specifier strings as well. All other `SET` features are PostgreSQL extensions.

See Also

The function `set_config` provides the equivalent capability. See *Miscellaneous Functions* in the *PostgreSQL User’s Guide*.

SET CONSTRAINTS

Name

`SET CONSTRAINTS` — set the constraint mode of the current transaction

Synopsis

```
SET CONSTRAINTS { ALL | constraint [, ...] } { DEFERRED | IMMEDIATE }
```

Description

`SET CONSTRAINTS` sets the behavior of constraint evaluation in the current transaction. In `IMMEDIATE` mode, constraints are checked at the end of each statement. In `DEFERRED` mode, constraints are not checked until transaction commit.

> **Note:** This command only alters the behavior of constraints within the current transaction. Thus, if you execute this command outside of an explicit transaction block (such as one started with `BEGIN`), it will not appear to have any effect. If you wish to change the behavior of a constraint without needing to issue a `SET CONSTRAINTS` command in every transaction, specify `INITIALLY DEFERRED` or `INITIALLY IMMEDIATE` when you create the constraint.

When you change the mode of a constraint to be `IMMEDIATE`, the new constraint mode takes effect retroactively: any outstanding data modifications that would have been checked at the end of the transaction (when using `DEFERRED`) are instead checked during the execution of the `SET CONSTRAINTS` command.

Upon creation, a constraint is always give one of three characteristics: `INITIALLY DEFERRED`, `INITIALLY IMMEDIATE DEFERRABLE`, or `INITIALLY IMMEDIATE NOT DEFERRABLE`. The third class is not affected by the `SET CONSTRAINTS` command.

Currently, only foreign key constraints are affected by this setting. Check and unique constraints are always effectively initially immediate not deferrable.

Compatibility

SQL92, SQL99

`SET CONSTRAINTS` is defined in SQL92 and SQL99. The implementation in PostgreSQL complies with the behavior defined in the standard, except for the PostgreSQL limitation that `SET CONSTRAINTS` cannot be applied to check or unique constraints.

SET SESSION AUTHORIZATION

Name

`SET SESSION AUTHORIZATION` — set the session user identifier and the current user identifier of the current session

Synopsis

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION username
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

Description

This command sets the session user identifier and the current user identifier of the current SQL-session context to be *`username`*. The user name may be written as either an identifier or a string literal. The session user identifier is valid for the duration of a connection; for example, it is possible to temporarily become an unprivileged user and later switch back to become a superuser.

The session user identifier is initially set to be the (possibly authenticated) user name provided by the client. The current user identifier is normally equal to the session user identifier, but may change temporarily in the context of "setuid" functions and similar mechanisms. The current user identifier is relevant for permission checking.

The session user identifier may be changed only if the initial session user (the *authenticated user*) had the superuser privilege. Otherwise, the command is accepted only if it specifies the authenticated user name.

The `SESSION` and `LOCAL` modifiers act the same as for the regular *SET* command.

The `DEFAULT` and `RESET` forms reset the session and current user identifiers to be the originally authenticated user name. These forms are always accepted.

Examples

```
SELECT SESSION_USER, CURRENT_USER;
 current_user | session_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;
 current_user | session_user
--------------+--------------
 paul         | paul
```

Compatibility

SQL99

SQL99 allows some other expressions to appear in place of the literal `*username*` which are not important in practice. PostgreSQL allows identifier syntax (`"username"`), which SQL does not. SQL does not allow this command during a transaction; PostgreSQL does not make this restriction because there is no reason to. The privileges necessary to execute this command are left implementation-defined by the standard.

SET TRANSACTION

Name

`SET TRANSACTION` — set the characteristics of the current transaction

Synopsis

```
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL
    { READ COMMITTED | SERIALIZABLE }
```

Description

This command sets the transaction isolation level. The `SET TRANSACTION` command sets the characteristics for the current SQL-transaction. It has no effect on any subsequent transactions. This command cannot be used after the first query or data-modification statement (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, `COPY`) of a transaction has been executed. `SET SESSION CHARACTERISTICS` sets the default transaction isolation level for each transaction for a session. `SET TRANSACTION` can override it for an individual transaction.

The isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently.

READ COMMITTED

A statement can only see rows committed before it began. This is the default.

SERIALIZABLE

The current transaction can only see rows committed before first query or data-modification statement was executed in this transaction.

> **Tip:** Intuitively, serializable means that two concurrent transactions will leave the database in the same state as if the two has been executed strictly after one another in either order.

Notes

The session default transaction isolation level can also be set with the command

```
SET default_transaction_isolation = 'value'
```

and in the configuration file. Consult the *Administrator's Guide* for more information.

Compatibility

SQL92, SQL99

`SERIALIZABLE` is the default level in SQL. PostgreSQL does not provide the isolation levels `READ UNCOMMITTED` and `REPEATABLE READ`. Because of multiversion concurrency control, the `SERIALIZABLE` level is not truly serializable. See the *User's Guide* for details.

In SQL there are two other transaction characteristics that can be set with these commands: whether the transaction is read-only and the size of the diagnostics area. Neither of these concepts are supported in PostgreSQL.

SHOW

Name

`SHOW` — show the value of a run-time parameter

Synopsis

```
SHOW name

SHOW ALL
```

Inputs

`name`

The name of a run-time parameter. See *SET* for a list.

ALL

Show all current session parameters.

Description

`SHOW` will display the current setting of a run-time parameter. These variables can be set using the `SET` statement, by editing the `postgresql.conf`, through the `PGOPTIONS` environmental variable, or through a command-line flag when starting the postmaster.

Even with `autocommit` set to `off`, `SHOW` does not start a new transaction block. See the `autocommit` section of the *Administrator's Guide* for details.

Diagnostics

`ERROR: Option 'name' is not recognized`

Message returned if *`name`* does not stand for an existing parameter.

Examples

Show the current `DateStyle` setting:

```
SHOW DateStyle;
               DateStyle
```

```
---------------------------------------
 ISO with US (NonEuropean) conventions
(1 row)
```

Show the current genetic optimizer (`geqo`) setting:

```
SHOW GEQO;
 geqo
------
 on
(1 row)
```

Show all settings:

```
SHOW ALL;
             name              |                setting
-------------------------------+---------------------------------------

 australian_timezones          | off
 authentication_timeout        | 60
 checkpoint_segments           | 3
    .
    .
    .
 wal_debug                     | 0
 wal_sync_method               | fdatasync
(94 rows)
```

Compatibility

The `SHOW` command is a PostgreSQL extension.

See Also

The function `current_setting` produces equivalent output. See *Miscellaneous Functions* in the *PostgreSQL User's Guide*.

START TRANSACTION

Name

`START TRANSACTION` — start a transaction block

Synopsis

```
    START TRANSACTION [ ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE } ]
```

Inputs

None.

Outputs

`START TRANSACTION`

Message returned if successful.

`WARNING: BEGIN: already a transaction in progress`

If there is already a transaction in progress when the command is issued.

Description

This command begins a new transaction. If the isolation level is specified, the new transaction has that isolation level. In all other respects, the behavior of this command is identical to the *BEGIN* command.

Notes

The isolation level of a transaction can also be set with the *SET TRANSACTION* command. If no isolation level is specified, the default isolation level is used.

Compatibility

SQL99

`SERIALIZABLE` is the default isolation level in SQL99, but it is not the usual default in PostgreSQL: the factory default setting is READ COMMITTED. PostgreSQL does not provide the isolation levels `READ UNCOMMITTED` and `REPEATABLE READ`. Because of lack of predicate locking, the `SERIALIZABLE` level is not truly serializable. See the *User's Guide* for details.

In SQL99 this statement can specify two other properties of the new transaction: whether the transaction is read-only and the size of the diagnostics area. Neither of these concepts are currently supported in PostgreSQL.

TRUNCATE

Name

`TRUNCATE` — empty a table

Synopsis

```
TRUNCATE [ TABLE ] name
```

Inputs

name

The name (optionally schema-qualified) of the table to be truncated.

Outputs

`TRUNCATE TABLE`

Message returned if the table is successfully truncated.

Description

`TRUNCATE` quickly removes all rows from a table. It has the same effect as an unqualified `DELETE` but since it does not actually scan the table it is faster. This is most useful on large tables.

`TRUNCATE` cannot be executed inside a transaction block (`BEGIN`/`COMMIT` pair), because there is no way to roll it back.

Usage

Truncate the table `bigtable`:

```
TRUNCATE TABLE bigtable;
```

Compatibility

SQL92

There is no `TRUNCATE` in SQL92.

UNLISTEN

Name

`UNLISTEN` — stop listening for a notification

Synopsis

```
UNLISTEN { notifyname | * }
```

Inputs

notifyname

Name of previously registered notify condition.

`*`

All current listen registrations for this backend are cleared.

Outputs

`UNLISTEN`

Acknowledgment that statement has executed.

Description

`UNLISTEN` is used to remove an existing `NOTIFY` registration. `UNLISTEN` cancels any existing registration of the current PostgreSQL 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

notifyname need not be a valid class name but can be any string valid as a name up to 64 characters long.

The backend does not complain if you unlisten something you were not listening for. Each backend will automatically execute `UNLISTEN *` when exiting.

Usage

To subscribe to an existing registration:

```
LISTEN virtual;
LISTEN
NOTIFY virtual;
NOTIFY
Asynchronous NOTIFY ’virtual’ from backend with pid ’8448’ received
```

Once `UNLISTEN` has been executed, further `NOTIFY` commands will be ignored:

```
UNLISTEN virtual;
UNLISTEN
NOTIFY virtual;
NOTIFY
-- notice no NOTIFY event is received
```

Compatibility

SQL92

There is no `UNLISTEN` in SQL92.

UPDATE

Name

`UPDATE` — update rows of a table

Synopsis

```
UPDATE [ ONLY ] table SET col = expression [, ...]
    [ FROM fromlist ]
    [ WHERE condition ]
```

Inputs

`table`

The name (optionally schema-qualified) of an existing table. If `ONLY` is specified, only that table is updated. If `ONLY` is not specified, the table and all its descendant tables (if any) are updated. `*` can be appended to the table name to indicate that descendant tables are to be scanned, but in the current version, this is the default behavior. (In releases before 7.1, `ONLY` was the default behavior.) The default can be altered by changing the `SQL_INHERITANCE` configuration option.

`column`

The name of a column in *`table`*.

`expression`

A valid expression or value to assign to column.

`fromlist`

A PostgreSQL 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 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 columns in the statement.

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.

By default UPDATE will update tuples in the table specified and all its sub-tables. If you wish to only update the specific table mentioned, you should use the ONLY clause.

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 the positioned UPDATE statement:

```
UPDATE table SET column = expression [, ...]
    WHERE CURRENT OF cursor
```

where *`cursor`* identifies an open cursor.

VACUUM

Name

`VACUUM` — garbage-collect and optionally analyze a database

Synopsis

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table [ (column [, ...] ) ] ]
```

Inputs

FULL

Selects “full” vacuum, which may reclaim more space, but takes much longer and exclusively locks the table.

FREEZE

Selects aggressive “freezing” of tuples.

VERBOSE

Prints a detailed vacuum activity report for each table.

ANALYZE

Updates statistics used by the optimizer to determine the most efficient way to execute a query.

`table`

The name (optionally schema-qualified) of a specific table to vacuum. Defaults to all tables in the current database.

`column`

The name of a specific column to analyze. Defaults to all columns.

Outputs

`VACUUM`

The command is complete.

`INFO: --Relation table--`

The report header for *`table`*.

```
INFO: 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.

`INFO: Index *index*: Pages 28; Tuples 1000: Deleted 3000. Elapsed 0/0 sec.`

The analysis for an index on the target table.

Description

`VACUUM` reclaims storage occupied by deleted tuples. In normal PostgreSQL operation, tuples that are deleted or obsoleted by UPDATE are not physically removed from their table; they remain present until a `VACUUM` is done. Therefore it's necessary to do `VACUUM` periodically, especially on frequently-updated tables.

With no parameter, `VACUUM` processes every table in the current database. With a parameter, `VACUUM` processes only that table.

`VACUUM ANALYZE` performs a `VACUUM` and then an `ANALYZE` for each selected table. This is a handy combination form for routine maintenance scripts. See *ANALYZE* for more details about its processing.

Plain `VACUUM` (without `FULL`) simply reclaims space and makes it available for re-use. This form of the command can operate in parallel with normal reading and writing of the table, as an exclusive lock is not obtained. `VACUUM FULL` does more extensive processing, including moving of tuples across blocks to try to compact the table to the minimum number of disk blocks. This form is much slower and requires an exclusive lock on each table while it is being processed.

`FREEZE` is a special-purpose option that causes tuples to be marked "frozen" as soon as possible, rather than waiting until they are quite old. If this is done when there are no other open transactions in the same database, then it is guaranteed that all tuples in the database are "frozen" and will not be subject to transaction ID wraparound problems, no matter how long the database is left un-vacuumed. `FREEZE` is not recommended for routine use. Its only intended usage is in connection with preparation of user-defined template databases, or other databases that are completely read-only and will not receive routine maintenance `VACUUM` operations. See the *Administrator's Guide* for details.

Notes

We recommend that active production databases be `VACUUM`-ed frequently (at least nightly), in order to remove expired rows. After adding or deleting a large number of records, it may be a good idea to issue a `VACUUM ANALYZE` command for the affected table. This will update the system catalogs with the results of all recent changes, and allow the PostgreSQL query optimizer to make better choices in planning user queries.

The `FULL` option is not recommended for routine use, but may be useful in special cases. An example is when you have deleted most of the rows in a table and would like the table to physically shrink to occupy less disk space. `VACUUM FULL` will usually shrink the table more than a plain `VACUUM` would.

Usage

The following is an example from running `VACUUM` on a table in the regression database:

```
regression=> VACUUM VERBOSE ANALYZE onek;
INFO:  --Relation onek--
INFO:  Index onek_unique1: Pages 14; Tuples 1000: Deleted 3000.
```

```
        CPU 0.00s/0.11u sec elapsed 0.12 sec.
INFO:  Index onek_unique2: Pages 16; Tuples 1000: Deleted 3000.
        CPU 0.00s/0.10u sec elapsed 0.10 sec.
INFO:  Index onek_hundred: Pages 13; Tuples 1000: Deleted 3000.
        CPU 0.00s/0.10u sec elapsed 0.10 sec.
INFO:  Index onek_stringu1: Pages 31; Tuples 1000: Deleted 3000.
        CPU 0.01s/0.09u sec elapsed 0.10 sec.
INFO:  Removed 3000 tuples in 70 pages.
        CPU 0.02s/0.04u sec elapsed 0.07 sec.
INFO:  Pages 94: Changed 0, Empty 0; Tup 1000: Vac 3000, Keep 0, UnUsed 0.
        Total CPU 0.05s/0.45u sec elapsed 0.59 sec.
INFO:  Analyzing onek
VACUUM
```

Compatibility

SQL92

There is no `VACUUM` statement in SQL92.

II. PostgreSQL Client Applications

This part contains reference information for PostgreSQL client applications and utilities. Not all of these commands are of general utility, some may require special privileges. The common feature of these applications is that they can be run on any host, independent of where the database server resides.

clusterdb

Name

`clusterdb` — cluster a PostgreSQL database

Synopsis

`clusterdb` [*`connection-options`*...] [--table | -t *`table`*] [*`dbname`*]
`clusterdb` [*`connection-options`*...] [--all | -a]

Description

clusterdb is a utility for reclustering tables in a PostgreSQL database. It finds tables that have previously been clustered, and clusters them again on the same index that was last used. Tables that have never been clustered are not touched.

clusterdb is a shell script wrapper around the backend command *CLUSTER* via the PostgreSQL interactive terminal psql. There is no effective difference between clustering databases via this or other methods. psql must be found by the script and a database server must be running at the targeted host. Also, any default settings and environment variables available to psql and the libpq front-end library do apply.

clusterdb might need to connect several times to the PostgreSQL server, asking for a password each time. It is convenient to have a `$HOME/.pgpass` file in such cases.

Options

clusterdb accepts the following command-line arguments:

`-a`
`--all`

Cluster all databases.

`[-d]` *`dbname`*
`[--dbname]` *`dbname`*

Specifies the name of the database to be clustered. If this is not specified and `-a` (or `--all`) is not used, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used.

`-e`
`--echo`

Echo the commands that clusterdb generates and sends to the server.

`-q`
`--quiet`

Do not display a response.

`-t` *`table`*
`--table` *`table`*

Clusters *`table`* only.

clusterdb also accepts the following command-line arguments for connection parameters:

`-h` *`host`*
`--host` *`host`*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *`port`*
`--port` *`port`*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *`username`*
`--username` *`username`*

User name to connect as

`-W`
`--password`

Force password prompt.

Diagnostics

`CLUSTER`

Everything went well.

`clusterdb: Cluster failed.`

Something went wrong. clusterdb is only a wrapper script. See *CLUSTER* and psql for a detailed discussion of error messages and potential problems. Note that this message may appear once per table to be clustered.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters.

Examples

To cluster the database `test`:

```
$ clusterdb test
```

To cluster a single table `foo` in a database named `xyzzy`:

```
$ clusterdb --table foo xyzzy
```

See Also

CLUSTER

createdb

Name

`createdb` — create a new PostgreSQL database

Synopsis

```
createdb [options...] [dbname] [description]
```

Description

createdb creates a new PostgreSQL database.

Normally, the database user who executes this command becomes the owner of the new database. However a different owner can be specified via the `-O` option, if the executing user has appropriate privileges.

createdb is a shell script wrapper around the SQL command *CREATE DATABASE* via the PostgreSQL interactive terminal psql. Thus, there is nothing special about creating databases via this or other methods. This means that the psql program must be found by the script and that a database server must be running at the targeted port. Also, any default settings and environment variables available to psql and the libpq front-end library will apply.

Options

createdb accepts the following command-line arguments:

`dbname`

Specifies the name of the database to be created. The name must be unique among all PostgreSQL databases in this installation. The default is to create a database with the same name as the current system user.

`description`

This optionally specifies a comment to be associated with the newly created database.

`-D` *`location`*
`--location` *`location`*

Specifies the alternative location for the database. See also initlocation.

`-e`
`--echo`

Echo the queries that createdb generates and sends to the server.

`-E` *`encoding`*
`--encoding` *`encoding`*

Specifies the character encoding scheme to be used in this database.

`-O` *`owner`*
`--owner` *`owner`*

Specifies the database user who will own the new database.

`-q`
`--quiet`

Do not display a response.

`-T` *`template`*
`--template` *`template`*

Specifies the template database from which to build this database.

The options `-D`, `-E`, `-O`, and `-T` correspond to options of the underlying SQL command *CREATE DATABASE*; see there for more information about them.

createdb also accepts the following command-line arguments for connection parameters:

`-h` *`host`*
`--host` *`host`*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *`port`*
`--port` *`port`*

Specifies the Internet TCP/IP port or the local Unix domain socket file extension on which the server is listening for connections.

`-U` *`username`*
`--username` *`username`*

User name to connect as

`-W`
`--password`

Force password prompt.

Diagnostics

`CREATE DATABASE`

The database was successfully created.

`createdb: Database creation failed.`

(Says it all.)

`createdb: Comment creation failed. (Database was created.)`

The comment/description for the database could not be created. The database itself will have been created already. You can use the SQL command `COMMENT ON DATABASE` to create the comment later on.

If there is an error condition, the backend error message will be displayed. See *CREATE DATABASE* and psql for possibilities.

Environment

`PGDATABASE`

If set, the name of the database to create, unless overridden on the command line.

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters. `PGUSER` also determines the name of the database to create, if it is not specified on the command line or by `PGDATABASE`.

Examples

To create the database `demo` using the default database server:

```
$ createdb demo
CREATE DATABASE
```

The response is the same as you would have gotten from running the `CREATE DATABASE` SQL command.

To create the database `demo` using the server on host `eden`, port 5000, using the `LATIN1` encoding scheme with a look at the underlying query:

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE "demo" WITH ENCODING = 'LATIN1'
CREATE DATABASE
```

See Also

dropdb, *CREATE DATABASE*

createelang

Name

`createlang` — define a new PostgreSQL procedural language

Synopsis

`createlang` [*`connection-options`*...] *`langname`* [*`dbname`*]
`createlang` [*`connection-options`*...] --list | -l *`dbname`*

Description

createlang is a utility for adding a new programming language to a PostgreSQL database. createlang can handle all the languages supplied in the default PostgreSQL distribution, but not languages provided by other parties.

Although backend programming languages can be added directly using several SQL commands, it is recommended to use createlang because it performs a number of checks and is much easier to use. See *CREATE LANGUAGE* for additional information.

Options

createlang accepts the following command-line arguments:

`langname`

Specifies the name of the procedural programming language to be defined.

`[-d]` *`dbname`*
`[--dbname]` *`dbname`*

Specifies to which database the language should be added. The default is to use the database with the same name as the current system user.

`-e`
`--echo`

Displays SQL commands as they are executed.

`-l`
`--list`

Shows a list of already installed languages in the target database (which must be specified).

`-L` *`directory`*

Specifies the directory in which the language interpreter is to be found. The directory is normally found automatically; this option is primarily for debugging purposes.

createlang also accepts the following command-line arguments for connection parameters:

`-h` *`host`*
`--host` *`host`*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *`port`*
`--port` *`port`*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *`username`*
`--username` *`username`*

User name to connect as

`-W`
`--password`

Force password prompt.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters.

Diagnostics

Most error messages are self-explanatory. If not, run createlang with the `--echo` option and see under the respective SQL command for details. Check also under psql for more possibilities.

Notes

Use droplang to remove a language.

createlang is a shell script that invokes psql several times. If you have things arranged so that a password prompt is required to connect, you will be prompted for a password several times.

Examples

To install `pltcl` into the database `template1`:

```
$ createlang pltcl template1
```

See Also

droplang, *CREATE LANGUAGE*

createuser

Name

`createuser` — define a new PostgreSQL user account

Synopsis

`createuser` [*`options`*...] [*`username`*]

Description

createuser creates a new PostgreSQL user. Only superusers (users with `usesuper` set in the `pg_shadow` table) can create new PostgreSQL users, so createuser must be invoked by someone who can connect as a PostgreSQL superuser.

Being a superuser also implies the ability to bypass access permission checks within the database, so superuserdom should not be granted lightly.

createuser is a shell script wrapper around the SQL command *CREATE USER* via the PostgreSQL interactive terminal psql. Thus, there is nothing special about creating users via this or other methods. This means that the psql application must be found by the script and that a database server must be running at the targeted host. Also, any default settings and environment variables used by psql and the libpq front-end library will apply.

Options

createuser accepts the following command-line arguments:

`username`

> Specifies the name of the PostgreSQL user to be created. This name must be unique among all PostgreSQL users.

`-a`
`--adduser`

> The new user is allowed to create other users. (Note: Actually, this makes the new user a *superuser*. The option is poorly named.)

`-A`
`--no-adduser`

> The new user is not allowed to create other users (i.e., the new user is a regular user, not a superuser).

`-d`
`--createdb`

> The new user is allowed to create databases.

`-D`
`--no-createdb`

> The new user is not allowed to create databases.

`-e`
`--echo`

Echo the queries that createuser generates and sends to the server.

`-E`
`--encrypted`

Encrypts the user's password stored in the database. If not specified, the default is used.

`-i` *`uid`*
`--sysid` *`uid`*

Allows you to pick a non-default user ID for the new user. This is not necessary, but some people like it.

`-N`
`--unencrypted`

Does not encrypt the user's password stored in the database. If not specified, the default is used.

`-P`
`--pwprompt`

If given, createuser will issue a prompt for the password of the new user. This is not necessary if you do not plan on using password authentication.

`-q`
`--quiet`

Do not display a response.

You will be prompted for a name and other missing information if it is not specified on the command line.

createuser also accepts the following command-line arguments for connection parameters:

`-h` *`host`*
`--host` *`host`*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *`port`*
`--port` *`port`*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *`username`*
`--username` *`username`*

User name to connect as (not the user name to create)

`-W`
`--password`

Force password prompt (to connect to the server, not for the password of the new user).

Environment

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

Diagnostics

`CREATE USER`

All is well.

`createuser: creation of user "`*`username`*`" failed`

Something went wrong. The user was not created.

If there is an error condition, the backend error message will be displayed. See *CREATE USER* and psql for possibilities.

Examples

To create a user `joe` on the default database server:

```
$ createuser joe
Is the new user allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
```

To create the same user `joe` using the server on host `eden`, port 5000, avoiding the prompts and taking a look at the underlying query:

```
$ createuser -p 5000 -h eden -D -A -e joe
CREATE USER "joe" NOCREATEDB NOCREATEUSER
CREATE USER
```

See Also

dropuser, *CREATE USER*

dropdb

Name

`dropdb` — remove a PostgreSQL database

Synopsis

`dropdb` [*`options`*...] *`dbname`*

Description

dropdb destroys an existing PostgreSQL database. The user who executes this command must be a database superuser or the owner of the database.

dropdb is a shell script wrapper around the SQL command *DROP DATABASE* via the PostgreSQL interactive terminal psql. Thus, there is nothing special about dropping databases via this or other methods. This means that the psql must be found by the script and that a database server is running at the targeted host. Also, any default settings and environment variables available to psql and the libpq front-end library do apply.

Options

dropdb accepts the following command-line arguments:

`dbname`

Specifies the name of the database to be removed. The database must be one of the existing PostgreSQL databases in this installation.

`-e`
`--echo`

Echo the queries that dropdb generates and sends to the server.

`-i`
`--interactive`

Issues a verification prompt before doing anything destructive.

`-q`
`--quiet`

Do not display a response.

createdb also accepts the following command-line arguments for connection parameters:

`-h` *`host`*
`--host` *`host`*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *`port`*
`--port` *`port`*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *`username`*
`--username` *`username`*

User name to connect as

`-W`
`--password`

Force password prompt.

Diagnostics

`DROP DATABASE`

The database was successfully removed.

`dropdb: Database removal failed.`

Something didn't work out.

If there is an error condition, the backend error message will be displayed. See *DROP DATABASE* and psql for possibilities.

Environment

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

Examples

To destroy the database `demo` on the default database server:

```
$ dropdb demo
DROP DATABASE
```

To destroy the database `demo` using the server on host `eden`, port 5000, with verification and a peek at the underlying query:

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
```

```
Are you sure? (y/n) y
    DROP DATABASE "demo"
DROP DATABASE
```

See Also

createdb, *DROP DATABASE*

droplang

Name

`droplang` — remove a PostgreSQL procedural language

Synopsis

`droplang` [*connection-options*...] *langname* [*dbname*]
`droplang` [*connection-options*...] --list | -l *dbname*

Description

droplang is a utility for removing an existing programming language from a PostgreSQL database. droplang can drop any procedural language, even those not supplied by the PostgreSQL distribution.

Although backend programming languages can be removed directly using several SQL commands, it is recommended to use droplang because it performs a number of checks and is much easier to use. See *DROP LANGUAGE* for more.

Options

droplang accepts the following command line arguments:

langname

Specifies the name of the backend programming language to be removed.

`[-d]` *dbname*
`[--dbname]` *dbname*

Specifies from which database the language should be removed. The default is to use the database with the same name as the current system user.

`-e`
`--echo`

Displays SQL commands as they are executed.

`-l`
`--list`

Shows a list of already installed languages in the target database (which must be specified).

droplang also accepts the following command line arguments for connection parameters:

`-h` *host*
`--host` *host*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *`port`*
`--port` *`port`*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *`username`*
`--username` *`username`*

User name to connect as

`-W`
`--password`

Force password prompt.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters.

Diagnostics

Most error messages are self-explanatory. If not, run droplang with the `--echo` option and see under the respective SQL command for details. Check also under psql for more possibilities.

Notes

Use createlang to add a language.

Examples

To remove `pltcl`:

```
$ droplang pltcl dbname
```

See Also

createlang, *DROP LANGUAGE*

dropuser

Name

`dropuser` — remove a PostgreSQL user account

Synopsis

`dropuser` [*`options`*...] [*`username`*]

Description

dropuser removes an existing PostgreSQL user *and* the databases which that user owned. Only users with `usesuper` set in the `pg_shadow` table can destroy PostgreSQL users.

dropuser is a shell script wrapper around the SQL command *DROP USER* via the PostgreSQL interactive terminal psql. Thus, there is nothing special about removing users via this or other methods. This means that the psql must be found by the script and that a database server is running at the targeted host. Also, any default settings and environment variables available to psql and the libpq front-end library do apply.

Options

dropuser accepts the following command-line arguments:

`username`

Specifies the name of the PostgreSQL user to be removed. This name must exist in the PostgreSQL installation. You will be prompted for a name if none is specified on the command line.

`-e`
`--echo`

Echo the queries that dropuser generates and sends to the server.

`-i`
`--interactive`

Prompt for confirmation before actually removing the user.

`-q`
`--quiet`

Do not display a response.

createuser also accepts the following command-line arguments for connection parameters:

`-h` *`host`*
`--host` *`host`*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *`port`*
`--port` *`port`*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *`username`*
`--username` *`username`*

User name to connect as (not the user name to drop)

`-W`
`--password`

Force password prompt (to connect to the server, not for the password of the user to be dropped).

Environment

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

Diagnostics

`DROP USER`

All is well.

`dropuser: deletion of user "`*`username`*`" failed`

Something went wrong. The user was not removed.

If there is an error condition, the backend error message will be displayed. See *DROP USER* and psql for possibilities.

Examples

To remove user `joe` from the default database server:

```
$ dropuser joe
DROP USER
```

To remove user `joe` using the postmaster on host `eden`, port 5000, with verification and a peek at the underlying query:

```
$ dropuser -p 5000 -h eden -i -e joe
User "joe" and any owned databases will be permanently deleted.
```

```
Are you sure? (y/n) y
    DROP USER "joe"
DROP USER
```

See Also

createuser, *DROP USER*

ecpg

Name

`ecpg` — embedded SQL C preprocessor

Synopsis

`ecpg` [*`option`*...] *`file`*...

Description

`ecpg` is the embedded SQL preprocessor for C programs. It converts C programs with embedded SQL statements to normal C code by replacing the SQL invocations with special function calls. The output files can then be processed with any C compiler tool chain.

`ecpg` will convert each input file given on the command line to the corresponding C output file. Input files preferrably have the extension `.pgc`, in which case the extension will be replaced by `.c` to determine the output file name. If the extension of the input file is not `.pgc`, then the output file name is computed by appending `.c` to the full file name. The output file name can also be overridden using the `-o` option.

This reference page does not describe the embedded SQL language. See the *PostgreSQL Programmer's Guide* for that.

Options

`ecpg` accepts the following command-line arguments:

`-c`

Automatically generate C code from SQL code. Currently, this works for `EXEC SQL TYPE`.

`-D` *`symbol`*

Define a C preprocessor symbol.

`-I` *`directory`*

Specify an additional include path, used to find files included via `EXEC SQL INCLUDE`. Defaults are `.` (current directory), `/usr/local/include`, the PostgreSQL include directory which is defined at compile time (default: `/usr/local/pgsql/include`), and `/usr/include`, in that order.

`-o` *`filename`*

Specifies that ecpg should write all its output to the given *`filename`*.

`-t`

Turn on autocommit of transactions. In this mode, each query is automatically committed unless it is inside an explicit transaction block. In the default mode, queries are committed only when `EXEC SQL COMMIT` is issued.

`-v`

Print additional information including the version and the include path.

`---help`

Show a brief summary of the command usage, then exit.

`--version`

Output version information, then exit.

Notes

When compiling the preprocessed C code files, the compiler needs to be able to find the ECPG header files in the PostgreSQL include directory. Therefore, one might have to use the `-I` option when invoking the compiler (e.g., `-I/usr/local/pgsql/include`).

Programs using C code with embedded SQL have to be linked against the `libecpg` library, for example using the flags `-L/usr/local/pgsql/lib -lecpg`.

The value of either of these directories that is appropriate for the installation can be found out using pg_config.

Examples

If you have an embedded SQL C source file named `prog1.pgc`, you can create an executable program using the following sequence of commands:

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

See Also

PostgreSQL Programmer's Guide for a more detailed description of the embedded SQL interface

pg_config

Name

`pg_config` — retrieve information about the installed version of PostgreSQL

Synopsis

`pg_config` {--bindir | --includedir | --includedir-server | --libdir | --pkglibdir | --configure | --version...}

Description

The pg_config utility prints configuration parameters of the currently installed version of PostgreSQL. It is intended, for example, to be used by software packages that want to interface to PostgreSQL to facilitate finding the required header files and libraries.

Options

To use pg_config, supply one or more of the following options:

`--bindir`

Print the location of user executables. Use this, for example, to find the psql program. This is normally also the location where the `pg_config` program resides.

`--includedir`

Print the location of C header files of the client interfaces.

`--includedir-server`

Print the location of C header files for server programming.

`--libdir`

Print the location of object code libraries.

`--pkglibdir`

Print the location of dynamically loadable modules, or where the server would search for them. (Other architecture-dependent data files may also be installed in this directory.)

`--configure`

Print the options that were given to the `configure` script when PostgreSQL was configured for building. This can be used to reproduce the identical configuration, or to find out with what options a binary package was built. (Note however that binary packages often contain vendor-specific custom patches.)

`--version`

Print the version of PostgreSQL and exit.

If more than one option (except for `--version`) is given, the information is printed in that order, one item per line.

Notes

The option `--includedir-server` is new in PostgreSQL 7.2. In prior releases, the server include files were installed in the same location as the client headers, which could be queried with the `--includedir`. To make your package handle both cases, try the newer option first and test the exit status to see whether it succeeded.

In releases prior to PostgreSQL 7.1, before the `pg_config` came to be, a method for finding the equivalent configuration information did not exist.

History

The `pg_config` utility first appeared in PostgreSQL 7.1.

See Also

PostgreSQL Programmer's Guide

pg_dump

Name

`pg_dump` — extract a PostgreSQL database into a script file or other archive file

Synopsis

`pg_dump` [*`options`*...] [*`dbname`*]

Description

pg_dump is a utility for saving a PostgreSQL database into a script or an archive file. The script files are in plain-text format and contain the SQL commands required to reconstruct the database to the state it was in at the time it was saved. To restore these scripts, use psql. They can be used to reconstruct the database even on other machines and other architectures, with some modifications even on other SQL database products.

Furthermore, there are alternative archive file formats that are meant to be used with pg_restore to rebuild the database, and they also allow pg_restore to be selective about what is restored, or even to reorder the items prior to being restored. The archive files are also designed to be portable across architectures.

pg_dump will save the information necessary to re-generate all user-defined types, functions, tables, indexes, 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.

When used with one of the archive file formats and combined with pg_restore, pg_dump provides a flexible archival and transfer mechanism. pg_dump can be used to backup an entire database, then pg_restore can be used to examine the archive and/or select which parts of the database are to be restored. The most flexible output file format is the “custom” format (`-Fc`). It allows for selection and reordering of all archived items, and is compressed by default. The `tar` format (`-Ft`) is not compressed and it is not possible to reorder data when loading, but it is otherwise quite flexible; moreover, it can be manipulated with other tools such as `tar`.

While running `pg_dump`, one should examine the output for any warnings (printed on standard error), especially in light of the limitations listed below.

pg_dump makes consistent backups even if the database is being used concurrently. pg_dump does not block other users accessing the database (readers or writers).

Options

The following command-line options are used to control the output format.

`dbname`

Specifies the name of the database to be dumped. If this is not specified, the environment variable `PGDATABASE` is used. If that is not set, the user name specified for the connection is used.

`-a`
`--data-only`

Dump only the data, not the schema (data definitions).

This option is only meaningful for the plain-text format. For the other formats, you may specify the option when you call `pg_restore`.

`-b`
`--blobs`

Include large objects in dump.

`-c`
`--clean`

Output commands to clean (drop) database objects prior to (the commands for) creating them.

This option is only meaningful for the plain-text format. For the other formats, you may specify the option when you call `pg_restore`.

`-C`
`--create`

Begin the output with a command to create the database itself and reconnect to the created database. (With a script of this form, it doesn't matter which database you connect to before running the script.)

This option is only meaningful for the plain-text format. For the other formats, you may specify the option when you call `pg_restore`.

`-d`
`--inserts`

Dump data as `INSERT` commands (rather than `COPY`). This will make restoration very slow, but it makes the archives more portable to other SQL database packages.

`-D`
`--column-inserts`
`--attribute-inserts`

Dump data as `INSERT` commands with explicit column names (`INSERT INTO` *`table`* (*`column`*, ...) `VALUES ...`). This will make restoration very slow, but it is necessary if you desire to rearrange column ordering.

`-f` *`file`*
`--file=`*`file`*

Send output to the specified file. If this is omitted, the standard output is used.

`-F` *`format`*
`--format=`*`format`*

Selects the format of the output. *`format`* can be one of the following:

p

Output a plain-text SQL script file (default)

t

Output a `tar` archive suitable for input into pg_restore. Using this archive format allows reordering and/or exclusion of schema elements at the time the database is restored. It is also possible to limit which data is reloaded at restore time.

c

Output a custom archive suitable for input into pg_restore. This is the most flexible format in that it allows reordering of data load as well as schema elements. This format is also compressed by default.

`-i`
`--ignore-version`

Ignore version mismatch between pg_dump and the database server.

pg_dump can handle databases from previous releases of PostgreSQL, but very old versions are not supported anymore (currently prior to 7.0). Use this option if you need to override the version check (and if pg_dump then fails, don't say you weren't warned).

`-o`
`--oids`

Dump object identifiers (OIDs) for every table. Use this option if your application references the OID columns in some way (e.g., in a foreign key constraint). Otherwise, this option should not be used.

`-O`
`--no-owner`

Do not output commands to set the object ownership to match the original database. Typically, pg_dump issues (psql-specific) `\connect` statements to set ownership of schema elements. See also under `-R` and `-X use-set-session-authorization`. Note that `-O` does not prevent all reconnections to the database, only the ones that are exclusively used for ownership adjustments.

This option is only meaningful for the plain-text format. For the other formats, you may specify the option when you call `pg_restore`.

`-R`
`--no-reconnect`

Prohibit pg_dump from outputting a script that would require reconnections to the database while being restored. An average restoration script usually has to reconnect several times as different users to set the original ownerships of the objects. This option is a rather blunt instrument because it makes pg_dump lose this ownership information, *unless* you use the `-X use-set-session-authorization` option.

One possible reason why reconnections during restore might not be desired is if the access to the database requires manual interaction (e.g., passwords).

This option is only meaningful for the plain-text format. For the other formats, you may specify the option when you call `pg_restore`.

`-s`
`--schema-only`

Dump only the schema (data definitions), no data.

`-S` *`username`*
`--superuser=`*`username`*

Specify the superuser user name to use when disabling triggers. This is only relevant if `--disable-triggers` is used. (Usually, it's better to specify `--use-set-session-authorization`, and then start the resulting script as superuser.)

`-t` *`table`*
`--table=`*`table`*

Dump data for *`table`* only.

`-v`
`--verbose`

Specifies verbose mode. This will cause pg_dump to print progress messages to standard error.

`-x`
`--no-privileges`
`--no-acl`

Prevent dumping of access privileges (grant/revoke commands).

`-X use-set-session-authorization`
`--use-set-session-authorization`

Normally, if a (plain-text mode) script generated by pg_dump must alter the current database user (e.g., to set correct object ownerships), it uses the psql `\connect` command. This command actually opens a new connection, which might require manual interaction (e.g., passwords). If you use the `-X use-set-session-authorization` option, then pg_dump will instead output *SET SESSION AUTHORIZATION* commands. This has the same effect, but it requires that the user restoring the database from the generated script be a database superuser. This option effectively overrides the `-R` option.

Since *SET SESSION AUTHORIZATION* is a standard SQL command, whereas `\connect` only works in psql, this option also enhances the theoretical portability of the output script.

This option is only meaningful for the plain-text format. For the other formats, you may specify the option when you call `pg_restore`.

`-X disable-triggers`
`--disable-triggers`

This option is only relevant when creating a data-only dump. It instructs pg_dump to include commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have referential integrity checks or other triggers on the tables that you do not want to invoke during data reload.

Presently, the commands emitted for `--disable-triggers` must be done as superuser. So, you should also specify a superuser name with `-S`, or preferably specify `--use-set-session-authorization` and then be careful to start the resulting script as a superuser. If you give neither option, the entire script must be run as superuser.

This option is only meaningful for the plain-text format. For the other formats, you may specify the option when you call `pg_restore`.

`-Z` *`0..9`*
`--compress=`*`0..9`*

Specify the compression level to use in archive formats that support compression (currently only the custom archive format supports compression).

The following command-line options control the database connection parameters.

`-h` *`host`*
`--host=`*`host`*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *`port`*
`--port=`*`port`*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections. The port number defaults to 5432, or the value of the `PGPORT` environment variable (if set).

`-U` *`username`*

Connect as the given user

`-W`

Force a password prompt. This should happen automatically if the server requires password authentication.

Long option forms are only available on some platforms.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

Diagnostics

```
Connection to database ’template1’ failed.
connectDBStart() -- connect() failed: No such file or directory
        Is the postmaster running locally
        and accepting connections on Unix socket ’/tmp/.s.PGSQL.5432’?
```

pg_dump could not attach to the PostgreSQL server on the specified host and port. If you see this message, ensure that the server is running on the proper host and that you have specified the proper port.

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.

Notes

If your installation has any local additions to the template1 database, be careful to restore the output of pg_dump into a truly empty database; otherwise you are likely to get errors due to duplicate definitions of the added objects. To make an empty database without any local additions, copy from template0 not template1, for example:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

pg_dump has a few limitations:

- When dumping a single table or as plain text, pg_dump does not handle large objects. Large objects must be dumped in their entirety using one of the binary archive formats.
- When doing a data only dump, pg_dump emits queries to disable triggers on user tables before inserting the data and queries to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs may be left in the wrong state.

Members of tar archives are limited to a size less than 8 GB. (This is an inherent limitation of the tar file format.) Therefore this format cannot be used if the textual representation of a table exceeds that size. The total size of a tar archive and any of the other output formats is not limited, except possibly by the operating system.

Examples

To dump a database:

```
$ pg_dump mydb > db.out
```

To reload this database:

```
$ psql -d database -f db.out
```

To dump a database called `mydb` that contains large objects to a `tar` file:

```
$ pg_dump -Ft -b mydb > db.tar
```

To reload this database (with large objects) to an existing database called `newdb`:

```
$ pg_restore -d newdb db.tar
```

History

The pg_dump utility first appeared in Postgres95 release `0.02`. The non-plain-text output formats were introduced in PostgreSQL release `7.1`.

See Also

pg_dumpall, pg_restore, psql, *PostgreSQL Administrator’s Guide*

pg_dumpall

Name

`pg_dumpall` — extract a PostgreSQL database cluster into a script file

Synopsis

`pg_dumpall` [*`options`*...]

Description

pg_dumpall is a utility for writing out ("dumping") all PostgreSQL databases of a cluster into one script file. The script file contains SQL commands that can be used as input to psql to restore the databases. It does this by calling pg_dump for each database in a cluster. pg_dumpall also dumps global objects that are common to all databases. (pg_dump does not save these objects.) This currently includes the information about database users and groups.

Thus, pg_dumpall is an integrated solution for backing up your databases. But note a limitation: it cannot dump "large objects", since pg_dump cannot dump such objects into text files. If you have databases containing large objects, they should be dumped using one of pg_dump's non-text output modes.

Since pg_dumpall reads tables from all databases you will most likely have to connect as a database superuser in order to produce a complete dump. Also you will need superuser privileges to execute the saved script in order to be allowed to add users and groups, and to create databases.

The SQL script will be written to the standard output. Shell operators should be used to redirect it into a file.

pg_dumpall might need to connect several times to the PostgreSQL server, asking for a password each time. It is convenient to have a `$HOME/.pgpass` file in such cases.

Options

The following command-line options are used to control the output format.

`-c`
`--clean`

Include SQL commands to clean (drop) the databases before recreating them.

`-d`
`--inserts`

Dump data as `INSERT` commands (rather than `COPY`). This will make restoration very slow, but it makes the output more portable to other RDBMS packages.

`-D`
`--column-inserts`
`--attribute-inserts`

Dump data as `INSERT` commands with explicit column names (`INSERT INTO` *`table`* (*`column`*`, ...) VALUES ...`). This will make restoration very slow, but it is necessary if you desire to rearrange column ordering.

`-g`
`--globals-only`

Dump only global objects (users and groups), no databases.

`-i`
`--ignore-version`

Ignore version mismatch between pg_dumpall and the database server.

pg_dumpall can handle databases from previous releases of PostgreSQL, but very old versions are not supported anymore (currently prior to 7.0). Use this option if you need to override the version check (and if pg_dumpall then fails, don't say you weren't warned).

`-o`
`--oids`

Dump object identifiers (OIDs) for every table. Use this option if your application references the OID columns in some way (e.g., in a foreign key constraint). Otherwise, this option should not be used.

`-v`
`--verbose`

Specifies verbose mode. This will cause pg_dumpall to print progress messages to standard error.

The following command-line options control the database connection parameters.

-h *`host`*

Specifies the host name of the machine on which the database server is running. If host begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

-p *`port`*

The port number on which the server is listening. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

-U *`username`*

Connect as the given user.

-W

Force a password prompt. This should happen automatically if the server requires password authentication.

Long options are only available on some platforms.

Environment

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters.

Notes

Since pg_dumpall calls pg_dump internally, some diagnostic messages will refer to pg_dump.

pg_dumpall will need to connect several times to the PostgreSQL server. If password authentication is configured, it will ask for a password each time. In that case it would be convenient to set up a password file.

Examples

To dump all databases:

```
$ pg_dumpall > db.out
```

To reload this database use, for example:

```
$ psql -f db.out template1
```

(It is not important to which database you connect here since the script file created by pg_dumpall will contain the appropriate commands to create and connect to the saved databases.)

See Also

pg_dump, psql. Check there for details on possible error conditions.

pg_restore

Name

`pg_restore` — restore a PostgreSQL database from an archive file created by pg_dump

Synopsis

`pg_restore` [*`options`*...]

Description

pg_restore is a utility for restoring a PostgreSQL database from an archive created by pg_dump in one of the non-plain-text formats. It will issue the commands necessary to re-generate all user-defined types, functions, tables, indexes, aggregates, and operators, as well as the data in the tables.

The archive files contain information for pg_restore to rebuild the database, but also allow pg_restore to be selective about what is restored, or even to reorder the items prior to being restored. The archive files are designed to be portable across architectures.

pg_restore can operate in two modes: If a database name is specified, the archive is restored directly into the database. Otherwise, a script containing the SQL commands necessary to rebuild the database is created (and written to a file or standard output), similar to the ones created by the pg_dump plain text format. Some of the options controlling the script output are therefore analogous to pg_dump options.

Obviously, pg_restore cannot restore information that is not present in the archive file; for instance, if the archive was made using the "dump data as `INSERT`s" option, pg_restore will not be able to load the data using `COPY` statements.

Options

`pg_restore` accepts the following command line arguments. (Long option forms are only available on some platforms.)

`archive-name`

Specifies the location of the archive file to be restored. If not specified, the standard input is used.

`-a`
`--data-only`

Restore only the data, no schema (definitions).

`-c`
`--clean`

Clean (drop) database objects before recreating them.

`-C`
`--create`

Create the database before restoring into it. (When this switch appears, the database named with `-d` is used only to issue the initial `CREATE DATABASE` command. All data is restored into the database name that appears in the archive.)

`-d` *`dbname`*
`--dbname=`*`dbname`*

Connect to database *`dbname`* and restore directly into the database. Large objects can only be restored by using a direct database connection.

`-f` *`filename`*
`--file=`*`filename`*

Specify output file for generated script, or for the listing when used with `-l`. Default is the standard output.

`-F` *`format`*
`--format=`*`format`*

Specify format of the archive. It is not necessary to specify the format, since pg_restore will determine the format automatically. If specified, it can be one of the following:

t

Archive is a `tar` archive. Using this archive format allows reordering and/or exclusion of schema elements at the time the database is restored. It is also possible to limit which data is reloaded at restore time.

c

Archive is in the custom format of pg_dump. This is the most flexible format in that it allows reordering of data load as well as schema elements. This format is also compressed by default.

`-i`
`--ignore-version`

Ignore database version checks.

`-I` *`index`*
`--index=`*`index`*

Restore definition for named *`index`* only.

`-l`
`--list`

List the contents of the archive. The output of this command can be used with the `-L` option to restrict and reorder the items that are restored.

`-L` *`list-file`*
`--use-list=`*`list-file`*

Restore elements in *`list-file`* only, and in the order they appear in the file. Lines can be moved and may also be commented out by placing a `;` at the start of the line.

`-N`
`--orig-order`

Restore items in the original dump order. By default pg_dump will dump items in an order convenient to pg_dump, then save the archive in a modified OID order. This option overrides the OID ordering.

`-o`
`--oid-order`

Restore items in the OID order. By default pg_dump will dump items in an order convenient to pg_dump, then save the archive in a modified OID order. This option enforces strict OID ordering.

`-O`
`--no-owner`

Prevent any attempt to restore original object ownership. Objects will be owned by the user name used to attach to the database.

`-P` *`function-name(argtype [, ...])`*
`--function=`*`function-name(argtype [, ...])`*

Specify a procedure or function to be restored.

`-r`
`--rearrange`

Restore items in modified OID order. By default pg_dump will dump items in an order convenient to pg_dump, then save the archive in a modified OID order. Most objects will be restored in OID order, but some things (e.g., rules and indexes) will be restored at the end of the process irrespective of their OIDs. This option is the default.

`-R`
`--no-reconnect`

While restoring an archive, pg_restore typically has to reconnect to the database several times with different user names to set the correct ownership of the created objects. If this is undesirable (e.g., because manual interaction (passwords) would be necessary for each reconnection), this option prevents pg_restore from issuing any reconnection requests. (A connection request while in plain text mode, not connected to a database, is made by putting out a psql `\connect` command.) However, this option is a rather blunt instrument because it makes pg_restore lose all object ownership information, *unless* you use the `-X use-set-session-authorization` option.

`-s`
`--schema-only`

Restore the schema (definitions), no data. Sequence values will be reset.

`-S` *`username`*
`--superuser=`*`username`*

Specify the superuser user name to use when disabling triggers. This is only relevant if `--disable-triggers` is used.

`-t` *`table`*
`--table=`*`table`*

Restore schema/data for *`table`* only.

`-T` *`trigger`*
`--trigger=`*`trigger`*

Restore definition of *`trigger`* only.

`-v`
`--verbose`

Specifies verbose mode.

`-x`
`--no-privileges`
`--no-acl`

Prevent restoration of access privileges (grant/revoke commands).

`-X use-set-session-authorization`
`--use-set-session-authorization`

Normally, if restoring an archive requires altering the current database user (e.g., to set correct object ownerships), a new connection to the database must be opened, which might require manual interaction (e.g., passwords). If you use the `-X use-set-session-authorization` option, then pg_restore will instead use the *SET SESSION AUTHORIZATION* command. This has the same effect, but it requires that the user restoring the archive is a database superuser. This option effectively overrides the `-R` option.

`-X disable-triggers`
`--disable-triggers`

This option is only relevant when performing a data-only restore. It instructs pg_restore to execute commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have referential integrity checks or other triggers on the tables that you do not want to invoke during data reload.

Presently, the commands emitted for `--disable-triggers` must be done as superuser. So, you should also specify a superuser name with `-S`, or preferably specify `--use-set-session-authorization` and run pg_restore as a PostgreSQL superuser.

`pg_restore` also accepts the following command line arguments for connection parameters:

`-h` *`host`*
`--host=`*`host`*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *`port`*
`--port=`*`port`*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections. The port number defaults to 5432, or the value of the `PGPORT` environment variable (if set).

`-U` *`username`*

Connect as the given user

`-W`

Force a password prompt. This should happen automatically if the server requires password authentication.

Environment

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters.

Diagnostics

```
Connection to database ’template1’ failed.
connectDBStart() -- connect() failed: No such file or directory
        Is the postmaster running locally
        and accepting connections on Unix socket ’/tmp/.s.PGSQL.5432’?
```

pg_restore could not attach to the PostgreSQL server process on the specified host and port. If you see this message, ensure that the server 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.

Note: When a direct database connection is specified using the -d option, pg_restore internally executes SQL statements. If you have problems running pg_restore, make sure you are able to select information from the database using, for example, psql.

Notes

If your installation has any local additions to the `template1` database, be careful to load the output of pg_restore into a truly empty database; otherwise you are likely to get errors due to duplicate definitions of the added objects. To make an empty database without any local additions, copy from template0 not template1, for example:

```
CREATE DATABASE foo WITH TEMPLATE = template0;
```

The limitations of pg_restore are detailed below.

- When restoring data to a pre-existing table, pg_restore emits queries to disable triggers on user tables before inserting the data then emits queries to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs may be left in the wrong state.
- pg_restore will not restore large objects for a single table. If an archive contains large objects, then all large objects will be restored.

See also the pg_dump documentation for details on limitations of pg_dump.

Examples

To dump a database:

```
$ pg_dump mydb > db.out
```

To reload this database:

```
$ psql -d database -f db.out
```

To dump a database called `mydb` that contains large objects to a `tar` file:

```
$ pg_dump -Ft -b mydb > db.tar
```

To reload this database (with large objects) to an existing database called `newdb`:

```
$ pg_restore -d newdb db.tar
```

To reorder database items, it is first necessary to dump the table of contents of the archive:

```
$ pg_restore -l archive.file > archive.list
```

The listing file consists of a header and one line for each item, e.g.,

```
;
; Archive created at Fri Jul 28 22:28:36 2000
;     dbname: birds
;     TOC Entries: 74
;     Compression: 0
;     Dump Version: 1.4-0
;     Format: CUSTOM
;
;
; Selected TOC Entries:
;
2; 145344 TABLE species postgres
3; 145344 ACL species
4; 145359 TABLE nt_header postgres
5; 145359 ACL nt_header
6; 145402 TABLE species_records postgres
7; 145402 ACL species_records
8; 145416 TABLE ss_old postgres
9; 145416 ACL ss_old
10; 145433 TABLE map_resolutions postgres
11; 145433 ACL map_resolutions
12; 145443 TABLE hs_old postgres
13; 145443 ACL hs_old
```

Semi-colons are comment delimiters, and the numbers at the start of lines refer to the internal archive ID assigned to each item.

Lines in the file can be commented out, deleted, and reordered. For example,

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

could be used as input to `pg_restore` and would only restore items 10 and 6, in that order.

```
$ pg_restore -L archive.list archive.file
```

History

The pg_restore utility first appeared in PostgreSQL 7.1.

See Also

pg_dump, pg_dumpall, psql, *PostgreSQL Administrator’s Guide*

psql

Name

`psql` — PostgreSQL interactive terminal

Synopsis

`psql` [*`options`*] [*`dbname`* [*`user`*]]

Description

psql is a terminal-based front-end to PostgreSQL. It enables you to type in queries interactively, issue them to PostgreSQL, and see the query results. Alternatively, input can be from a file. In addition, it provides a number of meta-commands and various shell-like features to facilitate writing scripts and automating a wide variety of tasks.

Options

`-a`
`--echo-all`

Print all the lines to the screen as they are read. This is more useful for script processing rather than interactive mode. This is equivalent to setting the variable `ECHO` to `all`.

`-A`
`--no-align`

Switches to unaligned output mode. (The default output mode is otherwise aligned.)

`-c` *`query`*
`--command` *`query`*

Specifies that psql is to execute one query string, *`query`*, and then exit. This is useful in shell scripts.

`query` must be either a query string that is completely parsable by the backend (i.e., it contains no psql specific features), or it is a single backslash command. Thus you cannot mix SQL and psql meta-commands. To achieve that, you could pipe the string into psql, like this: `echo "\x \\ select * from foo;" | psql`.

`-d` *`dbname`*
`--dbname` *`dbname`*

Specifies the name of the database to connect to. This is equivalent to specifying *`dbname`* as the first non-option argument on the command line.

`-e`
`--echo-queries`

Show all queries that are sent to the backend. This is equivalent to setting the variable `ECHO` to `queries`.

`-E`
`--echo-hidden`

Echoes the actual queries generated by \d and other backslash commands. You can use this if you wish to include similar functionality into your own programs. This is equivalent to setting the variable `ECHO_HIDDEN` from within psql.

`-f` *`filename`*
`--file` *`filename`*

Use the file *`filename`* as the source of queries instead of reading queries interactively. After the file is processed, psql terminates. This is in many ways equivalent to the internal command `\i`.

If *`filename`* is `-` (hyphen), then standard input is read.

Using this option is subtly different from writing `psql <` *`filename`*. In general, both will do what you expect, but using `-f` enables some nice features such as error messages with line numbers. There is also a slight chance that using this option will reduce the start-up overhead. On the other hand, the variant using the shell's input redirection is (in theory) guaranteed to yield exactly the same output that you would have gotten had you entered everything by hand.

`-F` *`separator`*
`--field-separator` *`separator`*

Use *`separator`* as the field separator. This is equivalent to `\pset fieldsep` or `\f`.

`-h` *`hostname`*
`--host` *`hostname`*

Specifies the host name of the machine on which the postmaster is running. If host begins with a slash, it is used as the directory for the Unix-domain socket.

`-H`
`--html`

Turns on HTML tabular output. This is equivalent to `\pset format html` or the `\H` command.

`-l`
`--list`

Lists all available databases, then exits. Other non-connection options are ignored. This is similar to the internal command `\list`.

`-o` *`filename`*
`--output` *`filename`*

Put all query output into file *`filename`*. This is equivalent to the command `\o`.

`-p` *`port`*
`--port` *`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 or, if not set, to the port specified at compile time, usually 5432.

`-P` *`assignment`*
`--pset` *`assignment`*

Allows you to specify printing options in the style of `\pset` on the command line. Note that here you have to separate name and value with an equal sign instead of a space. Thus to set the output format to LaTeX, you could write `-P format=latex`.

`-q`
`--quiet`

Specifies that psql should do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this happens. This is useful with the `-c` option. Within psql you can also set the `QUIET` variable to achieve the same effect.

`-R` *`separator`*
`--record-separator` *`separator`*

Use *`separator`* as the record separator. This is equivalent to the `\pset recordsep` command.

`-s`
`--single-step`

Run in single-step mode. That means the user is prompted before each query is sent to the backend, with the option to cancel execution as well. Use this to debug scripts.

`-S`
`--single-line`

Runs in single-line mode where a newline terminates a query, as a semicolon does.

> **Note:** This mode is provided for those who insist on it, but you are not necessarily encouraged to use it. In particular, if you mix SQL and meta-commands on a line the order of execution might not always be clear to the inexperienced user.

`-t`
`--tuples-only`

Turn off printing of column names and result row count footers, etc. It is completely equivalent to the `\t` meta-command.

`-T` *`table_options`*
`--table-attr` *`table_options`*

Allows you to specify options to be placed within the HTML `table` tag. See `\pset` for details.

`-u`

Makes psql prompt for the user name and password before connecting to the database.

This option is deprecated, as it is conceptually flawed. (Prompting for a non-default user name and prompting for a password because the backend requires it are really two different things.) You are encouraged to look at the `-U` and `-W` options instead.

`-U` *`username`*
`--username` *`username`*

Connects to the database as the user *`username`* instead of the default. (You must have permission to do so, of course.)

`-v` *`assignment`*
`--set` *`assignment`*
`--variable` *`assignment`*

Performs a variable assignment, like the `\set` internal command. Note that you must separate name and value, if any, by an equal sign on the command line. To unset a variable, leave off the equal sign. To just set a variable without a value, use the equal sign but leave off the value. These assignments are done during a very early stage of start-up, so variables reserved for internal purposes might get overwritten later.

`-V`
`--version`

Shows the psql version.

`-W`
`--password`

Requests that psql should prompt for a password before connecting to a database. This will remain set for the entire session, even if you change the database connection with the meta-command `\connect`.

In the current version, psql automatically issues a password prompt whenever the backend requests password authentication. Because this is currently based on a hack, the automatic recognition might mysteriously fail, hence this option to force a prompt. If no password prompt is issued and the backend requires password authentication the connection attempt will fail.

`-x`
`--expanded`

Turns on extended row format mode. This is equivalent to the command `\x`.

`-X,`
`--no-psqlrc`

Do not read the start-up file `~/.psqlrc`.

`-?`
`--help`

Shows help about psql command line arguments.

Long options are not available on all platforms.

Exit Status

psql returns 0 to the shell if it finished normally, 1 if a fatal error of its own (out of memory, file not found) occurs, 2 if the connection to the backend went bad and the session is not interactive, and 3 if an error occurred in a script and the variable `ON_ERROR_STOP` was set.

Usage

Connecting To A Database

psql is a regular PostgreSQL client application. In order to connect to a database you need to know the name of your target database, the host name and port number of the server and what user name you want to connect as. psql can be told about those parameters via command line options, namely `-d`, `-h`, `-p`, and `-U` respectively. If an argument is found that does not belong to any option it will be interpreted as the database name (or the user name, if the database name is also given). Not all these options are required, defaults do apply. If you omit the host name, psql will connect via a Unix domain socket to a server on the local host. The default port number is compile-time determined. Since the database server uses the same default, you will not have to specify the port in most cases. The default user name is your Unix user name, as is the default database name. Note that you can't just connect to any database under any user name. Your database administrator should have informed you about your access rights. To save you some typing you can also set the environment variables `PGDATABASE`, `PGHOST`, `PGPORT` and `PGUSER` to appropriate values.

If the connection could not be made for any reason (e.g., insufficient privileges, postmaster is not running on the server, etc.), psql will return an error and terminate.

Entering Queries

In normal operation, psql provides a prompt with the name of the database to which psql is currently connected, followed by the string =>. For example,

```
$ psql testdb
Welcome to psql 7.3.2, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

testdb=>
```

At the prompt, the user may type in SQL queries. Ordinarily, input lines are sent to the backend when a query-terminating semicolon is reached. An end of line does not terminate a query! Thus queries can be spread over several lines for clarity. If the query was sent and without error, the query results are displayed on the screen.

Whenever a query is executed, psql also polls for asynchronous notification events generated by *LISTEN* and *NOTIFY*.

Meta-Commands

Anything you enter in psql that begins with an unquoted backslash is a psql meta-command that is processed by psql itself. These commands are what makes psql interesting for administration or scripting. Meta-commands are more commonly called slash or backslash 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 whitespace characters.

To include whitespace into an argument you may quote it with a single quote. To include a single quote into such an argument, precede it by a backslash. Anything contained in single quotes is furthermore subject to C-like substitutions for `\n` (new line), `\t` (tab), `\`*`digits`*, `\0`*`digits`*, and `\0x`*`digits`* (the character with the given decimal, octal, or hexadecimal code).

If an unquoted argument begins with a colon (`:`), it is taken as a psql variable and the value of the variable is used as the argument instead.

Arguments that are enclosed in backquotes (`‘`) are taken as a command line that is passed to the shell. The output of the command (with any trailing newline removed) is taken as the argument value. The above escape sequences also apply in backquotes.

Some commands take an SQL identifier (such as a table name) as argument. These arguments follow the syntax rules of SQL regarding double quotes: an identifier without double quotes is coerced to lower-case, while whitespace within double quotes is included in the argument.

Parsing for arguments stops when another unquoted backslash occurs. This is taken as the beginning of a new meta-command. The special sequence `\\` (two backslashes) marks the end of arguments and continues parsing SQL queries, if any. That way SQL and psql commands can be freely mixed on a line. But in any case, the arguments of a meta-command cannot continue beyond the end of the line.

The following meta-commands are defined:

`\a`

If the current table output format is unaligned, switch to aligned. If it is not unaligned, set it to unaligned. This command is kept for backwards compatibility. See `\pset` for a general solution.

`\cd` [*`directory`*]

Change the current working directory to *`directory`*. Without argument, change to the current user's home directory.

> **Tip:** To print your current working directory, use `\!pwd`.

`\C` [*`title`*]

Set the title of any tables being printed as the result of a query or unset any such title. This command is equivalent to `\pset title` *`title`*. (The name of this command derives from "caption", as it was previously only used to set the caption in an HTML table.)

`\connect` (or `\c`) [*`dbname`* [*`username`*]]

Establishes a connection to a new database and/or under a user name. The previous connection is closed. If *`dbname`* is `-` the current database name is assumed.

If *`username`* is omitted the current user name is assumed.

As a special rule, `\connect` without any arguments will connect to the default database as the default user (as you would have gotten by starting psql without any arguments).

If the connection attempt failed (wrong user name, access denied, etc.), the previous connection will be kept if and only if psql is in interactive mode. When executing a non-interactive script, processing will immediately stop with an error. This distinction was chosen as a user convenience against typos on the one hand, and a safety mechanism that scripts are not accidentally acting on the wrong database on the other hand.

`\copy` *`table`* `[ (` *`column_list`* `) ] { from | to }` *`filename`* `| stdin | stdout [ with ] [ oids ] [ delimiter [as]` *`'character'`* `] [ null [as]` *`'string'`* `]`

Performs a frontend (client) copy. This is an operation that runs an SQL *COPY* command, but instead of the backend's reading or writing the specified file, psql reads or writes the file and routes the data between the backend and the local file system. This means that file accessibility and privileges are those of the local user, not the server, and no SQL superuser privileges are required.

The syntax of the command is similar to that of the SQL `COPY` command (see its description for the details). Note that, because of this, special parsing rules apply to the `\copy` command. In particular, the variable substitution rules and backslash escapes do not apply.

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 the other technique may be preferable.

Note: Note the difference in interpretation of `stdin` and `stdout` between frontend and backend copies: in a frontend copy these always refer to psql's input and output stream. On a backend copy `stdin` comes from wherever the `COPY` itself came from (for example, a script run with the `-f` option), and `stdout` refers to the query output stream (see `\o` meta-command below).

`\copyright`

Shows the copyright and distribution terms of PostgreSQL.

`\d` [*`pattern`*]

For each relation (table, view, index, or sequence) matching the *`pattern`*, show all columns, their types, and any special attributes such as `NOT NULL` or defaults, if any. Associated indexes, constraints, rules, and triggers are also shown, as is the view definition if the relation is a view. ("Matching the pattern" is defined below.)

The command form `\d+` is identical, but any comments associated with the table columns are shown as well.

Note: If `\d` is used without a *`pattern`* argument, it is equivalent to `\dtvs` which will show a list of all tables, views, and sequences. This is purely a convenience measure.

`\da` [*`pattern`*]

Lists all available aggregate functions, together with the data type they operate on. If *`pattern`* (a regular expression) is specified, only matching aggregates are shown.

`\dd` [*`pattern`*]

Shows the descriptions of objects matching the *`pattern`*, or of all visible objects if no argument is given. But in either case, only objects that have a description are listed. ("Object" covers aggregates, functions, operators, types, relations (tables, views, indexes, sequences, large objects), rules, and triggers.) For example:

```
=> \dd version
                     Object descriptions
   Schema   |  Name   |  Object  |        Description
------------+---------+----------+--------------------------
 pg_catalog | version | function | PostgreSQL version string
(1 row)
```

Descriptions for objects can be created with the `COMMENT ON` SQL command.

Note: PostgreSQL stores the object descriptions in the `pg_description` system table.

`\dD` [*`pattern`*]

Lists all available domains (derived types). If *`pattern`* is specified, only matching domains are shown.

`\df [ `*`pattern`*` ]`

Lists available functions, together with their argument and return types. If *`pattern`* is specified, only matching functions are shown. If the form `\df+` is used, additional information about each function, including language and description, is shown.

> **Note:** To reduce clutter, `\df` does not show data type I/O functions. This is implemented by ignoring functions that accept or return type `cstring`.

`\distvS [ `*`pattern`*` ]`

This is not the actual command name: the letters i, s, t, v, S stand for index, sequence, table, view, and system table, respectively. You can specify any or all of these letters, in any order, to obtain a listing of all the matching objects. The letter S restricts the listing to system objects; without S, only non-system objects are shown. If "+" is appended to the command name, each object is listed with its associated description, if any.

If a *`pattern`* is specified, only objects whose name matches the pattern are listed.

`\dl`

This is an alias for `\lo_list`, which shows a list of large objects.

`\do [ `*`pattern`*` ]`

Lists available operators with their operand and return types. If a *`pattern`* is specified, only operators whose name matches the pattern are listed.

`\dp` [*`pattern`*]

Produces a list of all available tables with their associated access permissions. If a *`pattern`* is specified, only tables whose name matches the pattern are listed.

The commands GRANT and REVOKE are used to set access permissions. See GRANT for more information.

`\dT [ `*`pattern`*` ]`

Lists all data types or only those that match *`pattern`*. The command form `\dT+` shows extra information.

`\du [ `*`pattern`*` ]`

Lists all database users, or only those that match *`pattern`*.

`\edit` (or `\e`) [*`filename`*]

If *`filename`* is specified, the file is edited; after the editor exits, its content is copied back to the query buffer. If no argument is given, the current query buffer is copied to a temporary file which is then edited in the same fashion.

The new query buffer is then re-parsed according to the normal rules of psql, where the whole buffer is treated as a single line. (Thus you cannot make scripts this way. Use `\i` for that.) This means also that if the query ends with (or rather contains) a semicolon, it is immediately executed. In other cases it will merely wait in the query buffer.

> **Tip:** psql searches the environment variables `PSQL_EDITOR`, `EDITOR`, and `VISUAL` (in that order) for an editor to use. If all of them are unset, `/bin/vi` is run.

`\echo` *`text`* [...]

Prints the arguments to the standard output, separated by one space and followed by a newline. This can be useful to intersperse information in the output of scripts. For example:

```
=> \echo ‘date‘
Tue Oct 26 21:40:57 CEST 1999
```

If the first argument is an unquoted `-n` the the trailing newline is not written.

> **Tip:** If you use the `\o` command to redirect your query output you may wish to use `\qecho` instead of this command.

`\encoding` [*`encoding`*]

Sets the client encoding, if you are using multibyte encodings. Without an argument, this command shows the current encoding.

`\f` [*`string`*]

Sets the field separator for unaligned query output. The default is pipe (`|`). See also `\pset` for a generic way of setting output options.

`\g` [{ *`filename`* | `|`*`command`* }]

Sends the current query input buffer to the backend and optionally saves the output in *`filename`* or pipes the output into a separate Unix shell to execute *`command`*. A bare `\g` is virtually equivalent to a semicolon. A `\g` with argument is a “one-shot” alternative to the `\o` command.

`\help` (or `\h`) [*`command`*]

Give syntax help on the specified SQL command. If *`command`* is not specified, then psql will list all the commands for which syntax help is available. If *`command`* is an asterisk (“*”), then syntax help on all SQL commands is shown.

> **Note:** To simplify typing, commands that consists of several words do not have to be quoted. Thus it is fine to type **`\help alter table`**.

`\H`

Turns on HTML query output format. If the HTML format is already on, it is switched back to the default aligned text format. This command is for compatibility and convenience, but see `\pset` about setting other output options.

`\i` *`filename`*

Reads input from the file *`filename`* and executes it as though it had been typed on the keyboard.

> **Note:** If you want to see the lines on the screen as they are read you must set the variable `ECHO` to `all`.

`\l` (or `\list`)

List all the databases in the server as well as their owners. Append a "+" to the command name to see any descriptions for the databases as well. If your PostgreSQL installation was compiled with multibyte encoding support, the encoding scheme of each database is shown as well.

`\lo_export` *`loid`* *`filename`*

Reads the large object with OID *`loid`* from the database and writes it to *`filename`*. Note that this is subtly different from the server function `lo_export`, which acts with the permissions of the user that the database server runs as and on the server's file system.

> **Tip:** Use `\lo_list` to find out the large object's OID.

> **Note:** See the description of the `LO_TRANSACTION` variable for important information concerning all large object operations.

`\lo_import` *`filename`* [*`comment`*]

Stores the file into a PostgreSQL "large object". Optionally, it associates the given comment with the object. Example:

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

The response indicates that the large object received object id 152801 which one ought to remember if one wants to access the object ever again. For that reason it is recommended to always associate a human-readable comment with every object. Those can then be seen with the `\lo_list` command.

Note that this command is subtly different from the server-side `lo_import` because it acts as the local user on the local file system, rather than the server's user and file system.

> **Note:** See the description of the `LO_TRANSACTION` variable for important information concerning all large object operations.

`\lo_list`

Shows a list of all PostgreSQL "large objects" currently stored in the database, along with any comments provided for them.

`\lo_unlink` *`loid`*

Deletes the large object with OID *`loid`* from the database.

> **Tip:** Use `\lo_list` to find out the large object's OID.

> **Note:** See the description of the `LO_TRANSACTION` variable for important information concerning all large object operations.

`\o [ {filename | |command} ]`

Saves future query results to the file *`filename`* or pipes future results into a separate Unix shell to execute *`command`*. If no arguments are specified, the query output will be reset to `stdout`.

"Query results" includes all tables, command responses, and notices obtained from the database server, as well as output of various backslash commands that query the database (such as `\d`), but not error messages.

> **Tip:** To intersperse text output in between query results, use `\qecho`.

`\p`

Print the current query buffer to the standard output.

`\pset parameter [ value ]`

This command sets options affecting the output of query result tables. *`parameter`* describes which option is to be set. The semantics of *`value`* depend thereon.

Adjustable printing options are:

`format`

Sets the output format to one of `unaligned`, `aligned`, `html`, or `latex`. Unique abbreviations are allowed. (That would mean one letter is enough.)

"Unaligned" writes all fields of a tuple on a line, separated by the currently active field separator. This is intended to create output that might be intended to be read in by other programs (tab-separated, comma-separated). "Aligned" mode is the standard, human-readable, nicely formatted text output that is default. The "HTML" and "LaTeX" modes put out tables that are intended to be included in documents using the respective mark-up language. They are not complete documents! (This might not be so dramatic in HTML, but in LaTeX you must have a complete document wrapper.)

`border`

The second argument must be a number. In general, the higher the number the more borders and lines the tables will have, but this depends on the particular format. In HTML mode, this will translate directly into the `border=...` attribute, in the others only values 0 (no border), 1 (internal dividing lines), and 2 (table frame) make sense.

`expanded` (or `x`)

Toggles between regular and expanded format. When expanded format is enabled, all output has two columns with the field name on the left and the data on the right. This mode is useful if the data wouldn't fit on the screen in the normal "horizontal" mode.

Expanded mode is supported by all four output modes.

`null`

The second argument is a string that should be printed whenever a field is null. The default is not to print anything, which can easily be mistaken for, say, an empty string. Thus, one might choose to write `\pset null '(null)'`.

`fieldsep`

Specifies the field separator to be used in unaligned output mode. That way one can create, for example, tab- or comma-separated output, which other programs might prefer. To set a tab as field separator, type `\pset fieldsep '\t'`. The default field separator is `'|'` (a "pipe" symbol).

`footer`

Toggles the display of the default footer `(x rows)`.

`recordsep`

Specifies the record (line) separator to use in unaligned output mode. The default is a newline character.

`tuples_only` (or `t`)

Toggles between tuples only and full display. Full display may show extra information such as column headers, titles, and various footers. In tuples only mode, only actual table data is shown.

`title` [*`text`*]

Sets the table title for any subsequently printed tables. This can be used to give your output descriptive tags. If no argument is given, the title is unset.

> **Note:** This formerly only affected HTML mode. You can now set titles in any output format.

`tableattr` (or `T`) [*`text`*]

Allows you to specify any attributes to be placed inside the HTML `table` tag. This could for example be `cellpadding` or `bgcolor`. Note that you probably don't want to specify `border` here, as that is already taken care of by `\pset border`.

`pager`

Toggles the use of a pager for query and psql help output. If the environment variable `PAGER` is set, the output is piped to the specified program. Otherwise a platform-dependent default (such as `more`) is used.

In any case, psql only uses the pager if it seems appropriate. That means among other things that the output is to a terminal and that the table would normally not fit on the screen. Because of the modular nature of the printing routines it is not always possible to predict the number of lines that will actually be printed. For that reason psql might not appear very discriminating about when to use the pager.

Illustrations on how these different formats look can be seen in the *Examples* section.

> **Tip:** There are various shortcut commands for `\pset`. See `\a`, `\C`, `\H`, `\t`, `\T`, and `\x`.

> **Note:** It is an error to call `\pset` without arguments. In the future this call might show the current status of all printing options.

`\q`

Quit the psql program.

`\qecho` *`text`* [...]

This command is identical to `\echo` except that all output will be written to the query output channel, as set by `\o`.

`\r`

Resets (clears) the query buffer.

`\s` [*`filename`*]

Print or save the command line history to *`filename`*. If *`filename`* is omitted, the history is written to the standard output. This option is only available if psql is configured to use the GNU history library.

> **Note:** In the current version, it is no longer necessary to save the command history, since that will be done automatically on program termination. The history is also loaded automatically every time psql starts up.

`\set` [*`name`* [*`value`* [...]]]

Sets the internal variable *`name`* to *`value`* or, if more than one value is given, to the concatenation of all of them. If no second argument is given, the variable is just set with no value. To unset a variable, use the `\unset` command.

Valid variable names can contain characters, digits, and underscores. See the section about psql variables for details.

Although you are welcome to set any variable to anything you want, psql treats several variables as special. They are documented in the section about variables.

> **Note:** This command is totally separate from the SQL command *SET*.

`\t`

Toggles the display of output column name headings and row count footer. This command is equivalent to `\pset tuples_only` and is provided for convenience.

`\T` *`table_options`*

Allows you to specify options to be placed within the `table` tag in HTML tabular output mode. This command is equivalent to `\pset tableattr` *`table_options`*.

`\timing`

Toggles a display of how long each SQL statement takes, in milliseconds.

`\w` {*`filename`* | `|`*`command`*}

Outputs the current query buffer to the file *`filename`* or pipes it to the Unix command *`command`*.

`\x`

Toggles extended row format mode. As such it is equivalent to `\pset expanded`.

`\z` [*`pattern`*]

Produces a list of all available tables with their associated access permissions. If a *`pattern`* is specified, only tables whose name matches the pattern are listed.

The commands GRANT and REVOKE are used to set access permissions. See GRANT for more information.

This is an alias for `\dp` ("display permissions").

`\!` [*`command`*]

Escapes to a separate Unix shell or executes the Unix command *`command`*. The arguments are not further interpreted, the shell will see them as is.

`\?`

Get help information about the backslash ("\") commands.

The various `\d` commands accept a *`pattern`* parameter to specify the object name(s) to be displayed. Patterns are interpreted similarly to SQL identifiers, in that unquoted letters are forced to lowercase, while double quotes (`"`) protect letters from case conversion and allow incorporation of whitespace into the identifier. Within double quotes, paired double quotes reduce to a single double quote in the resulting name. For example, `FOO"BAR"BAZ` is interpreted as `fooBARbaz`, and `"A weird"" name"` becomes `A weird" name`.

More interestingly, `\d` patterns allow the use of `*` to mean "any sequence of characters", and `?` to mean "any single character". (This notation is comparable to Unix shell filename patterns.) Advanced users can also use regular-expression notations such as character classes, for example `[0-9]` to match "any digit". To make any of these pattern-matching characters be interpreted literally, surround it with double quotes.

A pattern that contains an (unquoted) dot is interpreted as a schema name pattern followed by an object name pattern. For example, `\dt foo*.bar*` displays all tables in schemas whose name starts with `foo` and whose table name starts with `bar`. If no dot appears, then the pattern matches only objects that are visible in the current schema search path.

Whenever the *`pattern`* parameter is omitted completely, the `\d` commands display all objects that are visible in the current schema search path. To see all objects in the database, use the pattern `*.*`.

Advanced features

Variables

psql provides variable substitution features similar to common Unix command shells. This feature is new and not very sophisticated, yet, but there are plans to expand it in the future. Variables are simply name/value pairs, where the value can be any string of any length. To set variables, use the psql meta-command `\set`:

```
testdb=> \set foo bar
```

sets the variable "foo" to the value "bar". To retrieve the content of the variable, precede the name with a colon and use it as the argument of any slash command:

```
testdb=> \echo :foo
bar
```

> **Note:** The arguments of `\set` are subject to the same substitution rules as with other commands. Thus you can construct interesting references such as `\set :foo 'something'` and get "soft links" or "variable variables" of Perl or PHP fame, respectively. Unfortunately (or fortunately?), there is no way to do anything useful with these constructs. On the other hand, `\set bar :foo` is a perfectly valid way to copy a variable.

If you call `\set` without a second argument, the variable is simply set, but has no value. To unset (or delete) a variable, use the command `\unset`.

psql's internal variable names can consist of letters, numbers, and underscores in any order and any number of them. A number of regular variables are treated specially by psql. They indicate certain option settings that can be changed at run time by altering the value of the variable or represent some state of the application. Although you can use these variables for any other purpose, this is not recommended, as the program behavior might grow really strange really quickly. By convention, all specially treated variables consist of all upper-case letters (and possibly numbers and underscores). To ensure maximum compatibility in the future, avoid such variables. A list of all specially treated variables follows.

`DBNAME`

The name of the database you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

`ECHO`

If set to "`all`", all lines entered or from a script are written to the standard output before they are parsed or executed. To specify this on program start-up, use the switch `-a`. If set to "`queries`", psql merely prints all queries as they are sent to the backend. The option for this is `-e`.

`ECHO_HIDDEN`

When this variable is set and a backslash command queries the database, the query is first shown. This way you can study the PostgreSQL internals and provide similar functionality in your own programs. If you set the variable to the value `noexec`, the queries are just shown but are not actually sent to the backend and executed.

`ENCODING`

The current client multibyte encoding. If you are not set up to use multibyte characters, this variable will always contain "SQL_ASCII".

`HISTCONTROL`

If this variable is set to `ignorespace`, lines which begin with a space are not entered into the history list. If set to a value of `ignoredups`, lines matching the previous history line are not entered. A value of `ignoreboth` combines the two options. If unset, or if set to any other value than those above, all lines read in interactive mode are saved on the history list.

> **Note:** This feature was shamelessly plagiarized from bash.

`HISTSIZE`

The number of commands to store in the command history. The default value is 500.

> **Note:** This feature was shamelessly plagiarized from bash.

`HOST`

The database server host you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

`IGNOREEOF`

If unset, sending an EOF character (usually **Control**+**D**) to an interactive session of psql will terminate the application. If set to a numeric value, that many EOF characters are ignored before the application terminates. If the variable is set but has no numeric value, the default is 10.

> **Note:** This feature was shamelessly plagiarized from bash.

`LASTOID`

The value of the last affected OID, as returned from an `INSERT` or `lo_insert` command. This variable is only guaranteed to be valid until after the result of the next SQL command has been displayed.

`LO_TRANSACTION`

If you use the PostgreSQL large object interface to specially store data that does not fit into one tuple, all the operations must be contained in a transaction block. (See the documentation of the large object interface for more information.) Since psql has no way to tell if you already have a transaction in progress when you call one of its internal commands (`\lo_export`, `\lo_import`, `\lo_unlink`) it must take some arbitrary action. This action could either be to roll back any transaction that might already be in progress, or to commit any such transaction, or to do nothing at all. In the last case you must provide your own `BEGIN TRANSACTION`/`COMMIT` block or the results will be unpredictable (usually resulting in the desired action's not being performed in any case).

To choose what you want to do you set this variable to one of "rollback", "commit", or "nothing". The default is to roll back the transaction. If you just want to load one or a few objects this is fine. However, if you intend to transfer many large objects, it might be advisable to provide one explicit transaction block around all commands.

`ON_ERROR_STOP`

By default, if non-interactive scripts encounter an error, such as a malformed SQL query or internal meta-command, processing continues. This has been the traditional behavior of psql but it is sometimes not desirable. If this variable is set, script processing will immediately terminate. If the script was called from another script it will terminate in the same fashion. If the outermost script was not called from an interactive psql session but rather using the `-f` option, psql will return error code 3, to distinguish this case from fatal error conditions (error code 1).

`PORT`

The database server port to which you are currently connected. This is set every time you connect to a database (including program start-up), but can be unset.

`PROMPT1`
`PROMPT2`
`PROMPT3`

These specify what the prompt psql issues is supposed to look like. See "*Prompting*" below.

`QUIET`

This variable is equivalent to the command line option `-q`. It is probably not too useful in interactive mode.

`SINGLELINE`

This variable is set by the command line option `-S`. You can unset or reset it at run time.

`SINGLESTEP`

This variable is equivalent to the command line option `-s`.

`USER`

The database user you are currently connected as. This is set every time you connect to a database (including program start-up), but can be unset.

SQL Interpolation

An additional useful feature of psql variables is that you can substitute (“interpolate”) them into regular SQL statements. The syntax for this is again to prepend the variable name with a colon (`:`).

```
testdb=> \set foo ’my_table’
testdb=> SELECT * FROM :foo;
```

would then query the table `my_table`. The value of the variable is copied literally, so it can even contain unbalanced quotes or backslash commands. You must make sure that it makes sense where you put it. Variable interpolation will not be performed into quoted SQL entities.

A popular application of this facility is to refer to the last inserted OID in subsequent statements to build a foreign key scenario. Another possible use of this mechanism is to copy the contents of a file into a field. First load the file into a variable and then proceed as above.

```
testdb=> \set content ’\” ‘cat my_file.txt‘ ’\”
testdb=> INSERT INTO my_table VALUES (:content);
```

One possible problem with this approach is that `my_file.txt` might contain single quotes. These need to be escaped so that they don’t cause a syntax error when the third line is processed. This could be done with the program sed:

```
testdb=> \set content ’\” ‘sed -e "s/’/\\\\\\’/g" < my_file.txt‘ ’\”
```

Observe the correct number of backslashes (6)! You can resolve it this way: After psql has parsed this line, it passes `sed -e "s/’/\\\’/g" < my_file.txt` to the shell. The shell will do its own thing inside the double quotes and execute `sed` with the arguments `-e` and `s/’/\\’/g`. When sed parses this it will replace the two backslashes with a single one and then do the substitution. Perhaps at one point you thought it was great that all Unix commands use the same escape character. And this is ignoring the fact that you might have to escape all backslashes as well because SQL text constants are also subject to certain interpretations. In that case you might be better off preparing the file externally.

Since colons may legally appear in queries, the following rule applies: If the variable is not set, the character sequence “colon+name” is not changed. In any case you can escape a colon with a backslash to protect it from interpretation. (The colon syntax for variables is standard SQL for embedded query languages, such as ecpg. The colon syntax for array slices and type casts are PostgreSQL extensions, hence the conflict.)

Prompting

The prompts psql issues can be customized to your preference. The three variables `PROMPT1`, `PROMPT2`, and `PROMPT3` contain strings and special escape sequences that describe the appearance of the prompt. Prompt 1 is the normal prompt that is issued when psql requests a new query. Prompt 2 is issued when more input is expected during query input because the query was not terminated with a semicolon or a quote was not closed. Prompt 3 is issued when you run an SQL `COPY` command and you are expected to type in the tuples on the terminal.

The value of the respective prompt variable is printed literally, except where a percent sign ("%") is encountered. Depending on the next character, certain other text is substituted instead. Defined substitutions are:

`%M`

The full host name (with domain name) of the database server, or `[local]` if the connection is over a Unix domain socket, or `[local:`*`/dir/name`*`]`, if the Unix domain socket is not at the compiled in default location.

`%m`

The host name of the database server, truncated after the first dot, or `[local]` if the connection is over a Unix domain socket.

`%>`

The port number at which the database server is listening.

`%n`

The user name you are connected as (not your local system user name).

`%/`

The name of the current database.

`%~`

Like `%/`, but the output is "~" (tilde) if the database is your default database.

`%#`

If the current user is a database superuser, then a "#", otherwise a ">".

`%R`

In prompt 1 normally "=", but "^" if in single-line mode, and "!" if the session is disconnected from the database (which can happen if `\connect` fails). In prompt 2 the sequence is replaced by "-", "*", a single quote, or a double quote, depending on whether psql expects more input because the query wasn't terminated yet, because you are inside a `/* ... */` comment, or because you are inside a quote. In prompt 3 the sequence doesn't resolve to anything.

`%`*`digits`*

If *`digits`* starts with `0x` the rest of the characters are interpreted as a hexadecimal digit and the character with the corresponding code is substituted. If the first digit is `0` the characters are interpreted as on octal number and the corresponding character is substituted. Otherwise a decimal number is assumed.

`%:`*`name`*`:`

The value of the psql, variable *`name`*. See the section "*Variables*" for details.

`%'`*`command`*`'`

The output of *`command`*, similar to ordinary "back-tick" substitution.

To insert a percent sign into your prompt, write `%%`. The default prompts are equivalent to `'%/%R%# '` for prompts 1 and 2, and `'>> '` for prompt 3.

> **Note:** This feature was shamelessly plagiarized from tcsh.

Command-Line Editing

psql supports the Readline library for convenient line editing and retrieval. The command history is stored in a file named `.psql_history` in your home directory and is reloaded when psql starts up. Tab-completion is also supported, although the completion logic makes no claim to be an SQL parser. When available, psql is automatically built to use these features. If for some reason you do not like the tab completion, you can turn if off by putting this in a file named `.inputrc` in your home directory:

```
$if psql
set disable-completion on
$endif
```

(This is not a psql but a readline feature. Read its documentation for further details.)

Environment

`HOME`

Directory for initialization file (`.psqlrc`) and command history file (`.psql_history`).

`PAGER`

If the query results do not fit on the screen, they are piped through this command. Typical values are `more` or `less`. The default is platform-dependent. The use of the pager can be disabled by using the `\pset` command.

`PGDATABASE`

Default database to connect to

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

`PSQL_EDITOR`
`EDITOR`
`VISUAL`

Editor used by the `\e` command. The variables are examined in the order listed; the first that is set is used.

`SHELL`

Command executed by the `\!` command.

`TMPDIR`

Directory for storing temporary files. The default is `/tmp`.

Files

- Before starting up, psql attempts to read and execute commands from the file `$HOME/.psqlrc`. It could be used to set up the client or the server to taste (using the `\set` and `SET` commands).
- The command-line history is stored in the file `$HOME/.psql_history`.

Notes

- In an earlier life psql allowed the first argument of a single-letter backslash command to start directly after the command, without intervening whitespace. For compatibility this is still supported to some extent, but I am not going to explain the details here as this use is discouraged. If you get strange messages, keep this in mind. For example

  ```
  testdb=> \foo
  Field separator is "oo",
  ```

  which is perhaps not what one would expect.
- psql only works smoothly with servers of the same version. That does not mean other combinations will fail outright, but subtle and not-so-subtle problems might come up. Backslash commands are particularly likely to fail if the server is of a different version.
- Pressing Control-C during a “copy in” (data sent to the server) doesn’t show the most ideal of behaviors. If you get a message such as “COPY state must be terminated first”, simply reset the connection by entering `\c - -`.

Examples

> **Note:** This section only shows a few examples specific to psql. If you want to learn SQL or get familiar with PostgreSQL, you might wish to read the Tutorial that is included in the distribution.

The first example shows how to spread a query over several lines of input. Notice the changing prompt:

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text
testdb-> );
CREATE
```

Now look at the table definition again:

```
testdb=> \d my_table
             Table "my_table"
 Attribute |  Type   |      Modifier
-----------+---------+--------------------
```

```
 first     | integer | not null default 0
 second    | text    |
```

At this point you decide to change the prompt to something more interesting:

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

Let's assume you have filled the table with data and want to take a look at it:

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

You can make this table look differently by using the `\pset` command:

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep ","
Field separator is ",".
peter@localhost testdb=> \pset tuples_only
Showing only tuples.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
```

```
three,3
four,4
```

Alternatively, use the short commands:

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

pgtclsh

Name

`pgtclsh` — PostgreSQL Tcl shell client

Synopsis

`pgtclsh` [*`filename`* [*`arguments`*...]]

Description

`pgtclsh` is a Tcl shell interface extended with PostgreSQL database access functions. (Essentially, it is `tclsh` with `libpgtcl` loaded.) Like with the regular Tcl shell, the first command line argument is a script file, any remaining arguments are passed to the script. If no script file is named, the shell is interactive.

A Tcl shell with Tk and PostgreSQL functions is available as pgtksh.

See Also

pgtksh, *PostgreSQL Programmer's Guide* (description of `libpgtcl`) , tclsh

pgtksh

Name

`pgtksh` — PostgreSQL Tcl/Tk shell client

Synopsis

`pgtksh` [*`filename`* [*`arguments`*...]]

Description

`pgtksh` is a Tcl/Tk shell interface extended with PostgreSQL database access functions. (Essentially, it is `wish` with `libpgtcl` loaded.) Like with `wish`, the regular Tcl/Tk shell, the first command line argument is a script file, any remaining arguments are passed to the script. Special options may be processed by the X Window System libraries instead. If no script file is named, the shell is interactive.

A plain Tcl shell with PostgreSQL functions is available as pgtclsh.

See Also

pgtclsh, *PostgreSQL Programmer's Guide* (description of `libpgtcl`) , tclsh , wish

vacuumdb

Name

`vacuumdb` — garbage-collect and analyze a PostgreSQL database

Synopsis

`vacuumdb` [*connection-options*...] [--full | -f] [--verbose | -v] [--analyze | -z] [--table | -t '*table* [(*column* [,...])]'] [*dbname*]
`vacuumdb` [*connection-options*...] [--all | -a] [--full | -f] [--verbose | -v] [--analyze | -z]

Description

vacuumdb is a utility for cleaning a PostgreSQL database. vacuumdb will also generate internal statistics used by the PostgreSQL query optimizer.

vacuumdb is a shell script wrapper around the backend command *VACUUM* via the PostgreSQL interactive terminal psql. There is no effective difference between vacuuming databases via this or other methods. psql must be found by the script and a database server must be running at the targeted host. Also, any default settings and environment variables available to psql and the libpq front-end library do apply.

vacuumdb might need to connect several times to the PostgreSQL server, asking for a password each time. It is convenient to have a `$HOME/.pgpass` file in such cases.

Options

vacuumdb accepts the following command-line arguments:

`[-d]` *dbname*
`[--dbname]` *dbname*

Specifies the name of the database to be cleaned or analyzed. If this is not specified and `-a` (or `--all`) is not used, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used.

`-a`
`--all`

Vacuum all databases.

`-e`
`--echo`

Echo the commands that vacuumdb generates and sends to the server.

`-f`
`--full`

Perform “full” vacuuming.

`-q`
`--quiet`

Do not display a response.

`-t` *`table`* `[ (`*`column`* `[,...]) ]`
`--table` *`table`* `[ (`*`column`* `[,...]) ]`

Clean or analyze *`table`* only. Column names may be specified only in conjunction with the `--analyze` option.

> **Tip:** If you specify columns to vacuum, you probably have to escape the parentheses from the shell.

`-v`
`--verbose`

Print detailed information during processing.

`-z`
`--analyze`

Calculate statistics for use by the optimizer.

vacuumdb also accepts the following command-line arguments for connection parameters:

`-h` *`host`*
`--host` *`host`*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *`port`*
`--port` *`port`*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *`username`*
`--username` *`username`*

User name to connect as

`-W`
`--password`

Force password prompt.

Diagnostics

`VACUUM`

Everything went well.

`vacuumdb: Vacuum failed.`

Something went wrong. vacuumdb is only a wrapper script. See *VACUUM* and psql for a detailed discussion of error messages and potential problems.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters.

Examples

To clean the database `test`:

```
$ vacuumdb test
```

To clean and analyze for the optimizer a database named `bigdb`:

```
$ vacuumdb --analyze bigdb
```

To clean a single table `foo` in a database named `xyzzy`, and analyze a single column `bar` of the table for the optimizer:

```
$ vacuumdb --analyze --verbose --table ’foo(bar)’ xyzzy
```

See Also

VACUUM

III. PostgreSQL Server Applications

This part contains reference information for PostgreSQL server applications and support utilities. These commands can only be run usefully on the host where the database server resides. Other utility programs are listed in Reference II, *PostgreSQL Client Applications*.

initdb

Name

`initdb` — create a new PostgreSQL database cluster

Synopsis

`initdb` [options...] --pgdata | -D *`directory`*

Description

`initdb` creates a new PostgreSQL database cluster (or database system). A database cluster is a collection of databases that are managed by a single server instance.

Creating a database system consists of creating the directories in which the database data will live, generating the shared catalog tables (tables that belong to the whole cluster rather than to any particular database), and creating the `template1` database. When you create a new database, everything in the `template1` database is copied. It contains catalog tables filled in for things like the built-in types.

`initdb` initializes the database cluster's default locale and character set encoding. Some locale categories are fixed for the lifetime of the cluster, so it is important to make the right choice when running `initdb`. Other locale categories can be changed later when the server is started. `initdb` will write those locale settings into the `postgresql.conf` configuration file so they are the default, but they can be changed by editing that file. To set the locale that `initdb` uses, see the description of the `--locale` option. The character set encoding can be set separately for each database as it is created. `initdb` determines the encoding for the `template1` database, which will serve as the default for all other databases. To alter the default encoding use the `--encoding` option.

`initdb` must be run as the user that will own the server process, because the server needs to have access to the files and directories that `initdb` creates. Since the server may not be run as root, you must not run `initdb` as root either. (It will in fact refuse to do so.)

Although `initdb` will attempt to create the specified data directory, often it won't have permission to do so, since the parent of the desired data directory is often a root-owned directory. To set up an arrangement like this, create an empty data directory as root, then use `chown` to hand over ownership of that directory to the database user account, then `su` to become the database user, and finally run `initdb` as the database user.

Options

`-D` *`directory`*
`--pgdata=`*`directory`*

This option specifies the directory where the database system should be stored. This is the only information required by `initdb`, but you can avoid writing it by setting the `PGDATA` environment variable, which can be convenient since the database server (`postmaster`) can find the database directory later by the same variable.

`-E` *`encoding`*
`--encoding=`*`encoding`*

Selects the encoding of the template database. This will also be the default encoding of any database you create later, unless you override it there. To use the encoding feature, you must have enabled it at build time, at which time you also select the default for this option.

`--locale=`*`locale`*

Sets the default locale for the database cluster. If this option is not specified, the locale is inherited from the environment that `initdb` runs in.

`--lc-collate=`*`locale`*
`--lc-ctype=`*`locale`*
`--lc-messages=`*`locale`*
`--lc-monetary=`*`locale`*
`--lc-numeric=`*`locale`*
`--lc-time=`*`locale`*

Like `--locale`, but only sets the locale in the specified category.

`-U` *`username`*
`--username=`*`username`*

Selects the user name of the database superuser. This defaults to the name of the effective user running `initdb`. It is really not important what the superuser's name is, but one might choose to keep the customary name postgres, even if the operating system user's name is different.

`-W`
`--pwprompt`

Makes `initdb` prompt for a password to give the database superuser. If you don't plan on using password authentication, this is not important. Otherwise you won't be able to use password authentication until you have a password set up.

Other, less commonly used, parameters are also available:

`-d`
`--debug`

Print debugging output from the bootstrap backend and a few other messages of lesser interest for the general public. The bootstrap backend is the program `initdb` uses to create the catalog tables. This option generates a tremendous amount of extremely boring output.

`-L` *`directory`*

Specifies where `initdb` should find its input files to initialize the database system. This is normally not necessary. You will be told if you need to specify their location explicitly.

`-n`
`--noclean`

By default, when `initdb` determines that an error prevented it from completely creating the database system, it removes any files it may have created before discovering that it can't finish the job. This option inhibits tidying-up and is thus useful for debugging.

Environment

`PGDATA`

Specifies the directory where the database system is to be stored; may be overridden using the `-D` option.

See Also

postgres, postmaster, *PostgreSQL Administrator's Guide*

initlocation

Name

`initlocation` — create a secondary PostgreSQL database storage area

Synopsis

`initlocation` *`directory`*

Description

initlocation creates a new PostgreSQL secondary database storage area. See the discussion under *CREATE DATABASE* about how to manage and use secondary storage areas. If the argument does not contain a slash and is not valid as a path, it is assumed to be an environment variable, which is referenced. See the examples at the end.

In order to use this command you must be logged in (using `su`, for example) as the database superuser.

Examples

To create a database in an alternate location, using an environment variable:

```
$ export PGDATA2=/opt/postgres/data
```

Stop and start postmaster so it sees the `PGDATA2` environment variable. The system must be configured so the postmaster sees `PGDATA2` every time it starts. Finally:

```
$ initlocation PGDATA2
$ createdb -D PGDATA2 testdb
```

Alternatively, if you allow absolute paths you could write:

```
$ initlocation /opt/postgres/data
$ createdb -D /opt/postgres/data/testdb testdb
```

See Also

PostgreSQL Administrator's Guide

ipcclean

Name

`ipcclean` — remove shared memory and semaphores from an aborted PostgreSQL server

Synopsis

```
ipcclean
```

Description

`ipcclean` removes all shared memory segments and semaphore sets owned by the current user. It is intended to be used for cleaning up after a crashed PostgreSQL server (postmaster). Note that immediately restarting the server will also clean up shared memory and semaphores, so this command is of little real utility.

Only the database administrator should execute this program as it can cause bizarre behavior (i.e., crashes) if run during multiuser execution. If this command is executed while a postmaster is running, the shared memory and semaphores allocated by the postmaster will be deleted. This will result in a general failure of the backend servers started by that postmaster.

Notes

This script is a hack, but in the many years since it was written, no one has come up with an equally effective and portable solution. Since the postmaster can now clean up by itself, it is unlikely that `ipcclean` will be improved upon in the future.

The script makes assumption about the format of output of the ipcs utility which may not be true across different operating systems. Therefore, it may not work on your particular OS.

pg_ctl

Name

`pg_ctl` — start, stop, or restart a PostgreSQL server

Synopsis

`pg_ctl` start [-w] [-s] [-D *`datadir`*] [-l *`filename`*] [-o *`options`*] [-p *`path`*]
`pg_ctl` stop [-W] [-s] [-D *`datadir`*] [-m s[mart] | f[ast] | i[mmediate]]
`pg_ctl` restart [-w] [-s] [-D *`datadir`*] [-m s[mart] | f[ast] | i[mmediate]] [-o *`options`*]
`pg_ctl` reload [-s] [-D *`datadir`*]
`pg_ctl` status [-D *`datadir`*]

Description

pg_ctl is a utility for starting, stopping, or restarting postmaster, the PostgreSQL backend server, or displaying the status of a running postmaster. Although the postmaster can be started manually, pg_ctl encapsulates tasks such as redirecting log output, properly detaching from the terminal and process group, and it provides convenient options for controlled shutdown.

In `start` mode, a new postmaster is launched. The server is started in the background, the standard input attached to `/dev/null`. The standard output and standard error are either appended to a log file, if the `-l` option is used, or are redirected to pg_ctl's standard output (not standard error). If no log file is chosen, the standard output of pg_ctl should be redirected to a file or piped to another process, for example a log rotating program, otherwise the postmaster will write its output the the controlling terminal (from the background) and will not leave the shell's process group.

In `stop` mode, the postmaster that is running in the specified data directory is shut down. Three different shutdown methods can be selected with the `-m` option: "Smart" mode waits for all the clients to disconnect. This is the default. "Fast" mode does not wait for clients to disconnect. All active transactions are rolled back and clients are forcibly disconnected, then the database is shut down. "Immediate" mode will abort all server processes without clean shutdown. This will lead to a recovery run on restart.

`restart` mode effectively executes a stop followed by a start. This allows the changing of postmaster command line options.

`reload` mode simply sends the postmaster a SIGHUP signal, causing it to reread its configuration files (`postgresql.conf`, `pg_hba.conf`, etc.). This allows changing of configuration-file options that do not require a complete restart to take effect.

`status` mode checks whether a postmaster is running and if so displays the PID and the command line options that were used to invoke it.

Options

-D *`datadir`*

Specifies the file system location of the database files. If this is omitted, the environment variable `PGDATA` is used.

-l *`filename`*

Append the server log output to *`filename`*. If the file does not exist, it is created. The umask is set to 077, so access to the log file from other users is disallowed by default.

-m *`mode`*

Specifies the shutdown mode. *`mode`* may be `smart`, `fast`, or `immediate`, or the first letter of one of these three.

-o *`options`*

Specifies options to be passed directly to postmaster.

The parameters are usually surrounded by single or double quotes to ensure that they are passed through as a group.

-p *`path`*

Specifies the location of the `postmaster` executable. By default the postmaster is taken from the same directory as `pg_ctl`, or failing that, the hard-wired installation directory. It is not necessary to use this option unless you are doing something unusual and get errors that the postmaster was not found.

-s

Only print errors, no informational messages.

-w

Wait for the start or shutdown to complete. Times out after 60 seconds. This is the default for shutdowns.

-W

Do not wait for start or shutdown to complete. This is the default for starts and restarts.

Environment

`PGDATA`

Default data direction location

For others, see postmaster.

Files

If the file `postmaster.opts.default` exists in the data directory, the contents of the file will be passed as options to the postmaster, unless overridden by the `-o` option.

Notes

Waiting for complete start is not a well-defined operation and may fail if access control is set up so that a local client cannot connect without manual interaction. It should be avoided.

Examples

Starting the postmaster

To start up a postmaster:

```
$ pg_ctl start
```

An example of starting the postmaster, blocking until the postmaster comes up is:

```
$ pg_ctl -w start
```

For a postmaster using port 5433, and running without `fsync`, use:

```
$ pg_ctl -o "-F -p 5433" start
```

Stopping the postmaster

```
$ pg_ctl stop
```

stops the postmaster. Using the `-m` switch allows one to control *how* the backend shuts down.

Restarting the postmaster

This is almost equivalent to stopping the postmaster and starting it again except that `pg_ctl` saves and reuses the command line options that were passed to the previously running instance. To restart the postmaster in the simplest form:

```
$ pg_ctl restart
```

To restart postmaster, waiting for it to shut down and to come up:

```
$ pg_ctl -w restart
```

To restart using port 5433 and disabling `fsync` after restarting:

```
$ pg_ctl -o "-F -p 5433" restart
```

Showing postmaster status

Here is a sample status output from pg_ctl:

```
    $ pg_ctl status
    pg_ctl: postmaster is running (pid: 13718)
Command line was:
```

```
/usr/local/pgsql/bin/postmaster '-D' '/usr/local/pgsql/data' '-p' '5433' '-B' '128'
```

This is the command line that would be invoked in restart mode.

See Also

postmaster, *PostgreSQL Administrator's Guide*

pg_controldata

Name

`pg_controldata` — display server-wide control information

Synopsis

`pg_controldata [`*`datadir`*`]`

Description

`pg_controldata` returns information initialized during initdb, such as the catalog version and server locale. It also shows information about write-ahead logging and checkpoint processing. This information is server-wide, and not specific to any one database.

This utility may only be run by the user who installed the server because it requires read access to the `datadir`. You can specify the data directory on the command line, or use the environment variable `PGDATA`.

Environment

`PGDATA`

Default data directory location

pg_resetxlog

Name

`pg_resetxlog` — reset write-ahead log and pg_control contents

Synopsis

`pg_resetxlog` [-f] [-n] [-o *`oid`*] [-x *`xid`*] [-l *`fileid`*,*`seg`*] *`datadir`*

Description

`pg_resetxlog` clears the write-ahead log and optionally resets some fields in the `pg_control` file. This function is sometimes needed if these files have become corrupted. It should be used only as a last resort, when the server will not start due to such corruption.

After running this command, it should be possible to start the server, but bear in mind that the database may contain inconsistent data due to partially-committed transactions. You should immediately dump your data, run initdb, and reload. After reload, check for inconsistencies and repair as needed.

This utility can only be run by the user who installed the server, because it requires read/write access to the `datadir`. For safety reasons, you must specify the data directory on the command line. `pg_resetxlog` does not use the environment variable `PGDATA`.

If `pg_resetxlog` complains that it cannot determine valid data for `pg_control`, you can force it to proceed anyway by specifying the `-f` (force) switch. In this case plausible values will be substituted for the missing data. Most of the fields can be expected to match, but manual assistance may be needed for the next OID, next transaction ID, WAL starting address, and database locale fields. The first three of these can be set using the switches discussed below. `pg_resetxlog`'s own environment is the source for its guess at the locale fields; take care that `LANG` and so forth match the environment that initdb was run in. If you are not able to determine correct values for all these fields, `-f` can still be used, but the recovered database must be treated with even more suspicion than usual --- an immediate dump and reload is imperative. *Do not* execute any data-modifying operations in the database before you dump, as any such action is likely to make the corruption worse.

The `-o`, `-x`, and `-l` switches allow the next OID, next transaction ID, and WAL starting address values to be set manually. These are only needed when `pg_resetxlog` is unable to determine appropriate values by reading `pg_control`. A safe value for the next transaction ID may be determined by looking for the largest file name in `$PGDATA/pg_clog`, adding one, and then multiplying by 1048576. Note that the file names are in hexadecimal. It is usually easiest to specify the switch value in hexadecimal too. For example, if `0011` is the largest entry in `pg_clog`, `-x 0x1200000` will work (five trailing zeroes provide the proper multiplier). The WAL starting address should be larger than any file number currently existing in `$PGDATA/pg_xlog`. These also are in hex, and have two parts. For example, if `000000FF0000003A` is the largest entry in `pg_xlog`, `-l 0xFF,0x3B` will work. There is no comparably easy way to determine a next OID that's beyond the largest one in the database, but fortunately it is not critical to get the next-OID setting right.

The `-n` (no operation) switch instructs `pg_resetxlog` to print the values reconstructed from `pg_control` and then exit without modifying anything. This is mainly a debugging tool, but may be useful as a sanity check before allowing `pg_resetxlog` to proceed for real.

Notes

This command must not be used when the postmaster is running. `pg_resetxlog` will refuse to start up if it finds a postmaster lock file in the `datadir`. If the postmaster crashed then a lock file may have been left behind; in that case you can remove the lock file to allow `pg_resetxlog` to run. But before you do so, make doubly certain that there is no postmaster nor any backend server process still alive.

postgres

Name

`postgres` — run a PostgreSQL server in single-user mode

Synopsis

`postgres` [-A 0 | 1] [-B *nbuffers*] [-c *name*=*value*] [-d *debug-level*] [-D *datadir*] [-e] [-E] [-f s | i | t | n | m | h] [-F] [-i] [-N] [-o *filename*] [-O] [-P] [-s | -t pa | pl | ex] [-S *sort-mem*] [-W *seconds*] [--*name*=*value*] *database*
`postgres` [-A 0 | 1] [-B *nbuffers*] [-c *name*=*value*] [-d *debug-level*] [-D *datadir*] [-e] [-f s | i | t | n | m | h] [-F] [-i] [-o *filename*] [-O] [-p *database*] [-P] [-s | -t pa | pl | ex] [-S *sort-mem*] [-v *protocol-version*] [-W *seconds*] [--*name*=*value*]

Description

The `postgres` executable is the actual PostgreSQL server process that processes queries. It is normally not called directly; instead a postmaster multiuser server is started.

The second form above is how postgres is invoked by the postmaster (only conceptually, since both `postmaster` and `postgres` are in fact the same program); it should not be invoked directly this way. The first form invokes the server directly in interactive single-user mode. The primary use for this mode is during bootstrapping by initdb. Sometimes it is used for debugging or disaster recovery.

When invoked in interactive mode from the shell, the user can enter queries and the results will be printed to the screen, but in a form that is more useful for developers than end users. But note that running a single-user backend is not truly suitable for debugging the server since no realistic interprocess communication and locking will happen.

When running a stand-alone backend, the session user will be set to the user with ID 1. This user does not actually have to exist, so a stand-alone backend can be used to manually recover from certain kinds of accidental damage to the system catalogs. Implicit superuser powers are granted to the user with ID 1 in stand-alone mode.

Options

When postgres is started by a postmaster then it inherits all options set by the latter. Additionally, postgres-specific options can be passed from the postmaster with the `-o` switch.

You can avoid having to type these options by setting up a configuration file. See the *Administrator's Guide* for details. Some (safe) options can also be set from the connecting client in an application-dependent way. For example, if the environment variable `PGOPTIONS` is set, then libpq-based clients will pass that string to the server, which will interpret it as postgres command-line options.

General Purpose

The options `-A`, `-B`, `-c`, `-d`, `-D`, `-F`, and `--name` have the same meanings as the postmaster except that `-d 0` prevents the debugging level of the postmaster from being propagated to the backend.

`-e`

Sets the default date style to “European”, which means that the “day before month” (rather than month before day) rule is used to interpret ambiguous date input, and that the day is printed before the month in certain date output formats. See the *PostgreSQL User’s Guide* for more information.

`-o` *`filename`*

Sends all debugging and error output to *`filename`*. If the backend is running under the postmaster, this option is ignored, and the stderr inherited from the postmaster is used.

`-P`

Ignore system indexes while scanning/updating system tuples. The `REINDEX` command for system tables/indexes requires this option to be used.

`-s`

Print time information and other statistics at the end of each query. This is useful for benchmarking or for use in tuning the number of buffers.

`-S` *`sort-mem`*

Specifies the amount of memory to be used by internal sorts and hashes before resorting to temporary disk files. The value is specified in kilobytes, and defaults to 512 kilobytes. Note that for a complex query, several sorts and/or hashes might be running in parallel, and each one will be allowed to use as much as *`sort-mem`* kilobytes before it starts to put data into temporary files.

Options for stand-alone mode

`database`

Specifies the name of the database to be accessed. If it is omitted it defaults to the user name.

`-E`

Echo all queries.

`-N`

Disables use of newline as a query delimiter.

Semi-internal Options

There are several other options that may be specified, used mainly for debugging purposes. These are listed here only for the use by PostgreSQL system developers. *Use of any of these options is highly discouraged.* Furthermore, any of these options may disappear or change in a future release without notice.

`-f { s | i | m | n | h }`

Forbids the use of particular scan and join methods: `s` and `i` disable sequential and index scans respectively, while `n`, `m`, and `h` disable nested-loop, merge and hash joins respectively.

> **Note:** Neither sequential scans nor nested-loop joins can be disabled completely; the `-fs` and `-fn` options simply discourage the optimizer from using those plan types if it has any

other alternative.

`-i`

Prevents query execution, but shows the plan tree.

`-O`

Allows the structure of system tables to be modified. This is used by initdb.

`-p` *`database`*

Indicates that this server has been started by a postmaster and makes different assumptions about buffer pool management, file descriptors, etc.

`-t pa[rser] | pl[anner] | e[xecutor]`

Print timing statistics for each query relating to each of the major system modules. This option cannot be used together with the `-s` option.

`-v` *`protocol`*

Specifies the version number of the frontend/backend protocol to be used for this particular session.

`-W` *`seconds`*

As soon as this option is encountered, the process sleeps for the specified amount of seconds. This gives developers time to attach a debugger to the backend process.

Environment

`PGDATA`

Default data direction location

For others, which have little influence during single-user mode, see postmaster.

Notes

To stop a running query use the `SIGINT` signal. To tell postgres to reread the config file, use a `SIGHUP` signal. The postmaster uses `SIGTERM` to tell a postgres process to quit normally and `SIGQUIT` to terminate without the normal cleanup. These *should not* be used by users.

Usage

Start a stand-alone backend with a command like

```
postgres -D $PGDATA other-options my_database
```

Provide the correct path to the database area with `-D`, or make sure that the environment variable `PGDATA` is set. Also specify the name of the particular database you want to work in.

Normally, the stand-alone backend treats newline as the command entry terminator; there is no intelligence about semicolons, as there is in psql. To continue a command across multiple lines, you must type backslash just before each newline except the last one.

But if you use the `-N` command line switch, then newline does not terminate command entry. The backend will read the standard input until the end-of-file (EOF) marker, then process the input as a single query string. Backslash-newline is not treated specially in this case.

To quit the session, type EOF (**Control**+**D**, usually). If you've used `-N`, two consecutive EOFs are needed to exit.

Note that the stand-alone backend does not provide sophisticated line-editing features (no command history, for example).

See Also

initdb, ipcclean, postmaster

postmaster

Name

`postmaster` — PostgreSQL multiuser database server

Synopsis

`postmaster` [-A 0 | 1] [-B *`nbuffers`*] [-c *`name`*=*`value`*] [-d *`debug-level`*] [-D *`datadir`*] [-F] [-h *`hostname`*] [-i] [-k *`directory`*] [-l] [-N *`max-connections`*] [-o *`extra-options`*] [-p *`port`*] [-S] [--*`name`*=*`value`*] [-n | -s]

Description

postmaster is the PostgreSQL multiuser database server. In order for a client application to access a database it connects (over a network or locally) to a running postmaster. The postmaster then starts a separate server process (“postgres”) to handle the connection. The postmaster also manages the communication among server processes.

By default the postmaster starts in the foreground and prints log messages to the standard output. In practical applications the postmaster should be started as a background process, perhaps at boot time.

One postmaster always manages the data from exactly one database cluster. A database cluster is a collection of databases that is stored at a common file system location. When the postmaster starts it needs to know the location of the database cluster files (“data area”). This is done with the `-D` invocation option or the `PGDATA` environment variable; there is no default. More than one postmaster process can run on a system at one time, as long as they use different data areas and different communication ports (see below). A data area is created with initdb.

Options

postmaster accepts the following command line arguments. For a detailed discussion of the options consult the *Administrator’s Guide*. You can also save typing most of these options by setting up a configuration file.

-A 0|1

Enables run-time assert checks, which is a debugging aid to detect programming mistakes. This is only available if it was enabled during compilation. If so, the default is on.

-B *`nbuffers`*

Sets the number of shared buffers for use by the server processes. This value defaults to 64 buffers, where each buffer is 8 kB.

-c *`name`*=*`value`*

Sets a named run-time parameter. Consult the *Administrator’s Guide* for a list and descriptions. Most of the other command line options are in fact short forms of such a parameter assignment. `-c` can appear multiple times to set multiple parameters.

-d *`debug-level`*

Sets the debug level. The higher this value is set, the more debugging output is written to the server log. Values are from 1 to 5.

-D *`datadir`*

Specifies the file system location of the data directory. See discussion above.

-F

Disables `fsync` calls for performance improvement, at the risk of data corruption in event of a system crash. This parameter corresponds to setting `fsync=false` in `postgresql.conf`. Read the detailed documentation before using this!

`--fsync=true` has the opposite effect of this option.

-h *`hostname`*

Specifies the TCP/IP host name or address on which the postmaster is to listen for connections from client applications. Defaults to listening on all configured addresses (including localhost).

-i

Allows clients to connect via TCP/IP (Internet domain) connections. Without this option, only local Unix domain socket connections are accepted. This option corresponds to setting `tcpip_socket=true` in `postgresql.conf`.

`--tcpip_socket=false` has the opposite effect of this option.

-k *`directory`*

Specifies the directory of the Unix-domain socket on which the postmaster is to listen for connections from client applications. The default is normally `/tmp`, but can be changed at build time.

-l

Enables secure connections using SSL. The `-i` option is also required. You must have compiled with SSL enabled to use this option.

-N *`max-connections`*

Sets the maximum number of client connections that this postmaster will accept. By default, this value is 32, but it can be set as high as your system will support. (Note that `-B` is required to be at least twice `-N`. See the *Administrator's Guide* for a discussion of system resource requirements for large numbers of client connections.)

-o *`extra-options`*

The command line-style options specified in *`extra-options`* are passed to all backend server processes started by this postmaster. See postgres for possibilities. If the option string contains any spaces, the entire string must be quoted.

-p *`port`*

Specifies the TCP/IP port or local Unix domain socket file extension on which the postmaster is to listen for connections from client applications. Defaults to the value of the `PGPORT` environment variable, or if `PGPORT` is not set, then defaults to the value established during compilation (normally 5432). If you specify a port other than the default port, then all client applications must specify the same port using either command-line options or `PGPORT`.

-S

Specifies that the postmaster process should start up in silent mode. That is, it will disassociate from the user's (controlling) terminal, start its own process group, and redirect its standard output and standard error to `/dev/null`.

Using this switch discards all logging output, which is probably not what you want, since it makes it very difficult to troubleshoot problems. See below for a better way to start the postmaster in the background.

`--silent_mode=false` has the opposite effect of this option.

--*`name`*=*`value`*

Sets a named run-time parameter; a shorter form of `-c`.

Two additional command line options are available for debugging problems that cause a backend to die abnormally. These options control the behavior of the postmaster in this situation, and *neither option is intended for use in ordinary operation*.

The ordinary strategy for this situation is to notify all other backends that they must terminate and then reinitialize the shared memory and semaphores. This is because an errant backend could have corrupted some shared state before terminating.

These special-case options are:

-n

postmaster will not reinitialize shared data structures. A knowledgeable system programmer can then use a debugger to examine shared memory and semaphore state.

-s

postmaster will stop all other backend processes by sending the signal `SIGSTOP`, but will not cause them to terminate. This permits system programmers to collect core dumps from all backend processes by hand.

Environment

`PGCLIENTENCODING`

Default character encoding used by clients. (The clients may override this individually.) This value can also be set in the configuration file.

`PGDATA`

Default data direction location

`PGDATASTYLE`

Default value of the `datestyle` run-time parameter. (The use of this environment variable is deprecated.)

`PGPORT`

Default port (preferably set in the configuration file)

`TZ`

Server time zone

others

Other environment variables may be used to designate alternative data storage locations. See the *Administrator's Guide* for more information.

Diagnostics

`semget: No space left on device`

If you see this message, you should run the ipcclean command. After doing so, try starting postmaster again. If this still doesn't work, you probably need to configure your kernel for shared memory and semaphores as described in the installation notes. If you run multiple instances of postmaster on a single host, or have a kernel with particularly small shared memory and/or semaphore limits, you may have to reconfigure your kernel to increase its shared memory or semaphore parameters.

> **Tip:** You may be able to postpone reconfiguring your kernel by decreasing `-B` to reduce the shared memory consumption of PostgreSQL, and/or by reducing `-N` to reduce the semaphore consumption.

`StreamServerPort: cannot bind to port`

If you see this message, you should make certain that there is no other postmaster process already running on the same port number. The easiest way to determine this is by using the command

```
$ ps ax | grep postmaster
```

or

```
$ ps -e | grep postmaster
```

depending on your system.

If you are sure that no other postmaster processes are running and you still get this error, try specifying a different port using the `-p` option. You may also get this error if you terminate the postmaster and immediately restart it using the same port; in this case, you must simply wait a few seconds until the operating system closes the port before trying again. Finally, you may get this error if you specify a port number that your operating system considers to be reserved. For example, many versions of Unix consider port numbers under 1024 to be *trusted* and only permit the Unix superuser to access them.

Notes

If at all possible, *do not* use `SIGKILL` to kill the postmaster. This will prevent postmaster from freeing the system resources (e.g., shared memory and semaphores) that it holds before terminating.

To terminate the postmaster normally, the signals `SIGTERM`, `SIGINT`, or `SIGQUIT` can be used. The first will wait for all clients to terminate before quitting, the second will forcefully disconnect all clients, and the third will quit immediately without proper shutdown, resulting in a recovery run during restart.

The utility command pg_ctl can be used to start and shut down the postmaster safely and comfortably.

The `--` options will not work on FreeBSD or OpenBSD. Use `-c` instead. This is a bug in the affected operating systems; a future release of PostgreSQL will provide a workaround if this is not fixed.

Examples

To start postmaster in the background using default values, type:

```
$ nohup postmaster >logfile 2>&1 </dev/null &
```

To start postmaster with a specific port:

```
$ postmaster -p 1234
```

This command will start up postmaster communicating through the port 1234. In order to connect to this postmaster using psql, you would need to run it as

```
$ psql -p 1234
```

or set the environment variable `PGPORT`:

```
$ export PGPORT=1234
$ psql
```

Named run-time parameters can be set in either of these styles:

```
$ postmaster -c sort_mem=1234
$ postmaster --sort-mem=1234
```

Either form overrides whatever setting might exist for `sort_mem` in `postgresql.conf`. Notice that underscores in parameter names can be written as either underscore or dash on the command line.

> **Tip:** Except for short-term experiments, it's probably better practice to edit the setting in `postgresql.conf` than to rely on a command-line switch to set a parameter.

See Also

initdb, pg_ctl