PostgreSQL 7.1 Reference Manual

The PostgreSQL Global Development Group

PostgreSQL 7.1 Reference Manual

by The PostgreSQL Global Development Group

Table of Contents

I. SQL Commands

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

ABORT

Name

ABORT   Aborts the current transaction

Synopsis

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

Inputs

None.

Outputs

`ROLLBACK`

Message returned if successful.

`NOTICE: 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 Postgres extension present for historical reasons. **ROLLBACK** is the SQL92 equivalent command.

ALTER GROUP

Name

ALTER GROUP — Add users to a group, 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` Modifies table properties

Synopsis

```
ALTER TABLE [ ONLY ] table [ * ]
    ADD [ COLUMN ] column type
ALTER TABLE [ ONLY ] table [ * ]
    ALTER [ COLUMN ] column { SET DEFAULT value | DROP DEFAULT }
ALTER TABLE table [ * ]
    RENAME [ COLUMN ] column TO newcolumn
ALTER TABLE table
    RENAME TO newtable
ALTER TABLE table
    ADD table constraint definition
ALTER TABLE table
        OWNER TO new owner
```

Inputs

table

The name of an existing table to alter.

column

Name of a new or existing column.

type

Type of the new column.

newcolumn

New name for an existing column.

newtable

New name for the table.

table constraint definition

New table constraint for the table

`New user`

The user name of the new owner of the table.

Outputs

`ALTER`

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. The `ADD COLUMN` form adds a new column to the table using the same syntax as *CREATE TABLE*. The `ALTER COLUMN` form allows you to set or remove the default for the column. Note that defaults only apply to newly inserted rows. The `RENAME` clause causes the name of a table or column to change without changing any of the data contained in the affected table. Thus, the table or column will remain of the same type and size after this command is executed. The ADD *`table constraint definition`* clause adds a new constraint to the table using the same syntax as *CREATE TABLE*. The OWNER clause chnages the owner of the table to the user *`new user`*.

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

Notes

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

In the current implementation, default and constraint clauses for the new column will be ignored. You can use the `SET DEFAULT` form of **ALTER TABLE** to set the default later. (You will also have to update the already existing rows to the new default value, using *UPDATE*.)

In the current implementation, only FOREIGN KEY constraints can be added to a table. To create or remove a unique constraint, create a unique index (see *CREATE INDEX*). To add check constraints you need to recreate and reload the table, using other parameters to the *CREATE TABLE* command.

You must own the table in order to change it. Renaming any part of the schema of a system catalog is not permitted. The *PostgreSQL User's Guide* has further information on inheritance.

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

Usage

To add a column of type VARCHAR to a table:

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

To rename an existing column:

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

To rename an existing table:

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

To add a foreign key constraint to a table:

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address)
REFERENCES addresses(address) MATCH FULL
```

Compatibility

SQL92

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

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

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

> Removes a table constraint (such as a check constraint, unique constraint, or foreign key constraint). To remove a unique constraint, drop a unique index. To remove other kinds of constraints you need to recreate and reload the table, using other parameters to the *CREATE TABLE* command.
>
> For example, to drop any constraints on a table `distributors`:
>
> ```
> CREATE TABLE temp AS SELECT * FROM distributors;
> DROP TABLE distributors;
> CREATE TABLE distributors AS SELECT * FROM temp;
> DROP TABLE temp;
> ```

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

Removes a column from a table. Currently, to remove an existing column the table must be recreated and reloaded:

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

The clauses to rename columns and tables are Postgres extensions from SQL92.

ALTER USER

Name

`ALTER USER` Modifies user account information

Synopsis

```
ALTER USER username
    [ WITH PASSWORD 'password' ]
    [ CREATEDB | NOCREATEDB ] [ CREATEUSER | NOCREATEUSER ]
    [ VALID UNTIL 'abstime' ]
```

Inputs

username

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

password

The new password to be used for this account.

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.

Outputs

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.

Description

ALTER USER is used to change the attributes of a user's Postgres account. Only a database superuser can change privileges and password expiration with this command. Ordinary users can only change their own password.

Use *CREATE USER* to create a new user and *DROP USER* to remove a user.

Usage

Change a user password:

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

Change a user's valid until date:

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

Change a user's valid until date, specifying that his 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

SQL92

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

BEGIN

Name

`BEGIN` Begins a transaction in chained mode

Synopsis

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

Inputs

WORK
TRANSACTION

Optional keywords. They have no effect.

Outputs

`BEGIN`

This signifies that a new transaction has been started.

`NOTICE: BEGIN: already a transaction in progress`

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

Description

By default, Postgres 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*, *ROLLBACK*, or execution abort. Statements in chained mode are executed much faster, because transaction start/commit requires significant CPU and disk activity. Execution of multiple statements inside a transaction is also required for consistency when changing several related tables.

The default transaction isolation level in Postgres is READ COMMITTED, where queries inside the transaction see only changes committed before query execution. So, you have to use **SET TRANSACTION ISOLATION LEVEL SERIALIZABLE** just after **BEGIN** if you need more rigorous transaction isolation. In SERIALIZABLE mode queries will see only changes committed before the entire transaction began (actually, before execution of the first DML statement in a serializable transaction).

If the transaction is committed, Postgres will ensure either that all updates are done or else that none of them are done. Transactions have the standard ACID (atomic, consistent, isolatable, and durable) property.

Notes

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

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

Usage

To begin a user transaction:

```
BEGIN WORK;
```

Compatibility

SQL92

BEGIN is a Postgres 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 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`

Message returned if the cursor is successfully closed.

`NOTICE PerformPortalClose: portal "cursor" not found`

This warning is given if *cursor* is not declared or has already been closed.

Description

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

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

Notes

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

Usage

Close the cursor liahona:

```
CLOSE liahona;
```

Compatibility

SQL92

CLOSE is fully compatible with SQL92.

CLUSTER

Name

`CLUSTER` Gives storage clustering advice to the server

Synopsis

```
CLUSTER indexname ON tablename
```

Inputs

indexname

The name of an index.

table

The name of a table.

Outputs

`CLUSTER`

The clustering was done successfully.

`ERROR: relation <tablerelation_number> inherits "table"`

* *This is not documented anywhere. It seems not to be possible to cluster a table that is inherited.*

`ERROR: Relation table does not exist!`

* *The specified relation was not shown in the error message, which contained a random string instead of the relation name.*

Description

CLUSTER instructs Postgres to cluster the table specified by *table* approximately 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. The clustering is static. In other words, as the table is updated, the changes are not clustered. No attempt is made to keep new instances or updated tuples clustered. If one wishes, one can re-cluster manually by issuing the command again.

Notes

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

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

Another place 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.

There are two ways to cluster data. The first is with the **CLUSTER** command, which reorders the original table with the ordering of the index you specify. This can be slow on large tables because the rows are fetched from the heap in index order, and if the heap table is unordered, the entries are on random pages, so there is one disk page retrieved for every row moved. Postgres has a cache, but the majority of a big table will not fit in the cache.

Another way to cluster data is to use

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

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

Usage

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

```
CLUSTER emp_ind ON emp;
```

Compatibility

SQL92

There is no **CLUSTER** statement in SQL92.

COMMENT

Name

`COMMENT` Add comment to an object

Synopsis

```
COMMENT ON
[
  [ DATABASE | INDEX | RULE | SEQUENCE | TABLE | TYPE | VIEW ]
  object_name |
  COLUMN table_name.column_name|
  AGGREGATE agg_name agg_type|
  FUNCTION func_name (arg1, arg2, ...)|
  OPERATOR op (leftoperand_type rightoperand_type) |
  TRIGGER trigger_name ON table_name
] IS 'text'
```

Inputs

object_name, table_name, column_name, agg_name, func_name, op, trigger_name

The name of the object to be be commented.

text

The comment to add.

Outputs

`COMMENT`

Message returned if the table is successfully commented.

Description

COMMENT adds a comment to an object that can be easily retrieved with **psql**'s **\dd**, **\d+**, or **\l+** commands. To remove a comment, use `NULL`. Comments are automatically dropped when the object is dropped.

Usage

Comment the table `mytable`:

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

Some more examples:

```
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee id';
COMMENT ON RULE my_rule IS 'Logs UPDATES of employee records';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON TABLE my_table IS 'Employee Information';
COMMENT ON TYPE my_type IS 'Complex Number support';
COMMENT ON VIEW my_view IS 'View of departmental costs';
COMMENT ON COLUMN my_table.my_field IS 'Employee ID number';
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes sample
variance';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman Numeral';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two text';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for R.I.';
```

Compatibility

SQL92

There is no **COMMENT** in SQL92.

COMMIT

Name

`COMMIT` Commits 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.

`NOTICE: 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` Copies data between files and tables

Synopsis

```
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' ]
```

Inputs

BINARY

Changes the behavior of field formatting, forcing all data to be stored or read in binary format rather than as text. The DELIMITERS and WITH NULL options are irrelevant for binary format.

`table`

The name of an existing table.

WITH OIDS

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

`filename`

The absolute Unix pathname of the input or output file.

`stdin`

Specifies that input comes from the client application.

`stdout`

Specifies that output goes to the client application.

`delimiter`

The 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 Postgres tables and standard file-system files. **COPY TO** copies the entire 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).

COPY instructs the Postgres backend to directly read from or write to a file. If a file name is specified, the file must be accessible to the backend and the name must be specified from the viewpoint of the backend. If `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

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 phrase USING DELIMITERS. Characters in data fields which happen to match the delimiter character will be backslash quoted. Note that the delimiter is always a single character. If multiple characters are specified in the delimiter string, only the first character is used.

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

COPY TO neither invokes rules nor acts on column defaults. It does invoke triggers and check constraints.

COPY stops operation at the first error. This should not lead to problems in the event of a **COPY FROM**, but the target relation will already have received earlier rows in a **COPY TO**. 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 Postgres user (the userid the backend runs as), not the client. **COPY** naming a file is only allowed to database superusers, since it allows writing on any file that the backend has privileges to write on.

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

The actual format for each instance is

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

Note that the end of each row is marked by a Unix-style newline ("\n"). **COPY FROM** will not behave as desired if given a file containing DOS- or Mac-style newlines.

The OID is emitted as the first column if WITH OIDS is specified.

If **COPY TO** is sending its output to standard output instead of a file, after the last row it will send a backslash ("\") and a period (".") followed by a newline. Similarly, if **COPY FROM** is reading from standard input, it will expect a backslash ("\") and a period (".") followed by a newline, as the first three characters on a line to denote end-of-file. However, **COPY FROM** will terminate correctly (followed by the backend itself) if the input connection is closed before this special end-of-file pattern is found.

The backslash character has other special meanings. A literal backslash character is represented as two consecutive backslashes ("\\"). A literal tab character is represented as a backslash and a tab. (If you are using something other than tab as the column delimiter, backslash that delimiter character to include it in data.) A literal newline character is represented as a backslash and a newline. When loading text data not generated by Postgres, you will need to convert backslash characters ("\") to double-backslashes ("\\") to ensure that they are loaded properly.

Binary Format

The file format used for **COPY BINARY** changed in Postgres 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 datatype. Exactly N bytes of data follow the typlen word.

-1

Field is a varlena datatype. 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 datatype 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 we someday allow OIDs to be optional.

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 USING DELIMITERS '|';
```

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

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

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

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

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.

CREATE AGGREGATE

Name

`CREATE AGGREGATE` Defines 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 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`

Message returned if the command completes successfully.

Description

CREATE AGGREGATE allows a user or programmer to extend Postgres functionality by defining new aggregate functions. Some aggregate functions for base types such as `min(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.

An aggregate function is identified by its name and input data type. Two aggregates can have the same name if they operate on different input types. To avoid confusion, do not make an ordinary function of the same name and input data type as an aggregate.

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
```

Postgres 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" in pg_proc, 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 NULLs.

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

CREATE CONSTRAINT TRIGGER

Name

`CREATE CONSTRAINT TRIGGER` Create a trigger to support a constraint

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

Table name of the triggering relation.

constraint

Actual onstraint specification.

attributes

Constraint attributes.

func(*args*)

Function to call as part of the trigger processing.

Outputs

CREATE CONSTRAINT

Message returned if successful.

Description

CREATE CONSTRAINT TRIGGER is used from inside of **CREATE/ALTER TABLE** and by pg_dump to create the special triggers for referential integrity.

It is not intended for general use.

CREATE DATABASE

Name

CREATE DATABASE Creates a new database

Synopsis

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

Inputs

name

The name of a database to create.

dbpath

An alternate filesystem 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 Postgres database. The creator becomes the owner of the new database.

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 Postgres. 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 Postgres 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. In particular, it is essential that the source database be idle (no data-altering transactions in progress) for the duration of the copying operation. CREATE DATABASE will check that no backend processes (other than itself) are connected to the source database at the start of the operation, but this does not guarantee that changes cannot be made while the copy proceeds. Therefore, we recommend that databases used as templates be treated as read-only.

Two useful flags exist in `pg_database` for each database: `datistemplate` and `datallowconn`. `datistemplate` may be set to indicate that a database is intended as a template for CREATE DATABASE. If this flag is set, the database may be cloned by any user with CREATEDB privileges; if it is not set, only superusers and the owner of the database may clone it. If `datallowconn` is false, then no new connections to that database will be allowed (but existing sessions are not killed simply by setting the flag false). The `template0` database is normally marked this way to prevent modification of it.

Usage

To create a new database:

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

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

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

$ psql olly
Welcome to 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 FUNCTION

Name

CREATE FUNCTION Defines a new function

Synopsis

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    AS definition
    LANGUAGE 'langname'
    [ WITH ( attribute [, ...] ) ]
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    AS obj_file , link_symbol
    LANGUAGE 'langname'
    [ WITH ( attribute [, ...] ) ]
```

Inputs

name

The name of a function to create.

ftype

The data type(s) of the function's arguments, if any. The input types may be base or complex types, or *opaque*. `Opaque` indicates that the function accepts arguments of a non-SQL type such as `char *`.

rtype

The return data type. The output type may be specified as a base type, complex type, `setof type`, or `opaque`. The `setof` modifier indicates that the function will return a set of items, rather than a single item.

attribute

An optional piece of information about the function, used for optimization. See below for details.

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.

langname

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

Outputs

`CREATE`

This is returned if the command completes successfully.

Description

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

Function Attributes

The following items may appear in the WITH clause:

iscachable

: `Iscachable` indicates that the function always returns the same result when given the same argument values (i.e., it does not do database lookups or otherwise use information not directly present in its parameter list). The optimizer uses `iscachable` to know whether it is safe to pre-evaluate a call of the function.

isstrict

: `isstrict` indicates that the function always returns NULL whenever any of its arguments are NULL. If this attribute is specified, the function is not executed when there are NULL arguments; instead a NULL result is assumed automatically. When `isstrict` is not specified, the function will be called for NULL inputs. It is then the function author's responsibility to check for NULLs if necessary and respond appropriately.

Notes

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

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

The full SQL92 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.

Postgres 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.

Usage

To create a simple SQL function:

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

     answer
--------
      1
```

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

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

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

This example creates a function that does type conversion between the user-defined type complex, and the internal type point. The function is implemented by a dynamically loaded object that was compiled from C source. For Postgres 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';
```

The C declaration of the function is:

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

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

        return p;
}
```

Compatibility

SQL92

CREATE FUNCTION is a Postgres language extension.

SQL/PSM

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

SQL/PSM **CREATE FUNCTION** has the following syntax:

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

CREATE GROUP

Name

`CREATE GROUP` Creates a new group

Synopsis

```
CREATE GROUP name
    [ WITH
     [ SYSID gid ]
     [ USER  username [, ...] ] ]
```

Inputs

name

The name of the group.

gid

The `SYSID` clause can be used to choose the Postgres 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 adminstrator'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` Constructs a secondary index

Synopsis

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

Inputs

UNIQUE

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

`index_name`

The name of the index to be created.

`table`

The name of the table to be indexed.

`acc_name`

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

BTREE

an implementation of Lehman-Yao high-concurrency btrees.

RTREE

implements standard rtrees using Guttman's quadratic split algorithm.

HASH

an implementation of Litwin's linear hashing.

`column`

The name of a column of the table.

`ops_name`

An associated operator class. See below for details.

`func_name`

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

Outputs

`CREATE`

The message returned if the index is successfully created.

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

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

Description

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

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

In the first syntax shown above, the key field(s) for the index are specified as column names. Multiple fields can be specified if the index access method supports multi-column 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 indices* 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.

Postgres provides btree, rtree and hash access methods for indices. The btree access method is an implementation of Lehman-Yao high-concurrency btrees. The rtree access method implements standard rtrees 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).

Use *DROP INDEX* to remove an index.

Notes

The Postgres query optimizer will consider using a btree index whenever an indexed attribute is involved in a comparison using one of: <, <=, =, >=, >

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

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

Currently, only the btree access method supports multi-column indexes. Up to 16 keys may be specified by default (this limit can be altered when building Postgres).

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 btree 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 rtree indices 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. If the field on which your rectangles lie is about 20,000 units square or larger, you should use `bigbox_ops`.

The following query shows all defined operator classes:

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

Usage

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

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

Compatibility

SQL92

CREATE INDEX is a Postgres language extension.

There is no **CREATE INDEX** command in SQL92.

CREATE LANGUAGE

Name

`CREATE LANGUAGE` Defines a new language for functions

Synopsis

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

Inputs

TRUSTED

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

langname

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

HANDLER *call_handler*

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

comment

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

Outputs

`CREATE`

This message is returned if the language is successfully created.

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

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

Description

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

Writing PL handlers

Note: In Postgres 7.1 and later, call handlers must adhere to the "version 1" function manager interface, not the old-style interface.

The call handler for a procedural language must be written in a compiled language such as C and registered with Postgres as a function taking no arguments and returning the `opaque` type, a placeholder for unspecified or undefined types. This prevents the call handler from being called directly as a function from queries. (However, arguments may be supplied in the actual call when a PL function in the language offered by the handler is to be executed.)

The call handler is called in the same way as any other function: it receives a pointer to a FunctionCallInfoData struct containing argument values and information about the called function, and it is expected to return a Datum result (and possibly set the `isnull` field of the FunctionCallInfoData struct, if it wishes to return an SQL NULL result). The difference between a call handler and an ordinary callee function is that the `flinfo->fn_oid` field of the FunctionCallInfoData struct will contain the OID of the PL function to be called, not of the call handler itself. The call handler must use this field to determine which function to execute. Also, the passed argument list has been set up according to the declaration of the target PL function, not of the call handler.

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

Often, the same function is called many times per SQL statement. A call handler can avoid repeated lookups of information about the called function by using the `flinfo->fn_extra` field. This will initially be NULL, but can be set by the call handler to point at information about the PL function. On subsequent calls, if `flinfo->fn_extra` is already non-NULL then it can be used and the information lookup step skipped. The call handler must be careful that `flinfo->fn_extra` is made to point at memory that will live at least until the end of the current query, since an FmgrInfo data structure could be kept that long. One way to do this is to allocate the extra data in the memory context specified by `flinfo->fn_mcxt`; such data will normally have the same lifespan as the FmgrInfo itself. But the handler could also choose to use a longer-lived context so that it can cache function definition information across queries.

When a PL function is invoked as a trigger, no explicit arguments are passed, but the FunctionCallInfoData's `context` field points at a TriggerData node, rather than being NULL as it is in a plain function call. A PL handler should provide mechanisms for PL functions to get at the trigger information.

Notes

Use **CREATE FUNCTION** to create a function.

Use **DROP LANGUAGE** to drop procedural languages.

Refer to the table `pg_language` for further information:

```
        Table "pg_language"
   Attribute   |  Type   | Modifier
---------------+---------+----------
 lanname       | name    |
 lanispl       | boolean |
 lanpltrusted  | boolean |
 lanplcallfoid | oid     |
 lancompiler   | text    |

   lanname   | lanispl | lanpltrusted | lanplcallfoid | lancompiler
-------------+---------+--------------+---------------+-------------
 internal    | f       | f            |             0 | n/a
 C           | f       | f            |             0 | /bin/cc
 sql         | f       | f            |             0 | postgres
```

The call handler for a procedural language must normally be written in C and registered as 'internal' or 'C' language, depending on whether it is linked into the backend or dynamically loaded. The call handler cannot use the old-style 'C' function interface.

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

Usage

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

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

PG_FUNCTION_INFO_V1(plsample_call_handler);

Datum
plsample_call_handler(PG_FUNCTION_ARGS)
{
     Datum          retval;

     if (CALLED_AS_TRIGGER(fcinfo))
     {
          /*
           * Called as a trigger procedure
           */
          TriggerData    *trigdata = (TriggerData *) fcinfo->context;

          retval = ...
     } else {
          /*
           * Called as a function
           */

          retval = ...
     }

     return retval;
}
```

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

The following commands then register the sample procedural language:

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

Compatibility

SQL92

CREATE LANGUAGE is a Postgres extension. There is no **CREATE LANGUAGE** statement in SQL92.

CREATE OPERATOR

Name

`CREATE OPERATOR` Defines a new user operator

Synopsis

```
CREATE OPERATOR name ( PROCEDURE = func_name
     [, LEFTARG = type1 ] [, RIGHTARG = type2 ]
     [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
     [, RESTRICT = res_proc ] [, JOIN = join_proc ]
     [, HASHES ] [, SORT1 = left_sort_op ] [, SORT2 = right_sort_op ] )
```

Inputs

name

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

func_name

The function used to implement this operator.

type1

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

type2

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.

`left_sort_op`

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

`right_sort_op`

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

Outputs

CREATE

Message returned if the operator is successfully created.

Description

CREATE OPERATOR defines a new operator, *name*. The user who defines an operator becomes its owner.

The operator *name* is a sequence of up to NAMEDATALEN-1 (31 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 multi-character operator name.

"--" and "/*" cannot appear anywhere in an operator name, since they will be taken as the start of a comment.

A multi-character 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 Postgres 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 Postgres 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 Postgres 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, Postgres searches for it in the catalog. If it is found and it does not yet have a commutator itself, then the commutator's entry is updated to have the 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 Postgres 6.5, it also works to just have both operators refer to each other.)

The HASHES, SORT1, and SORT2 options are present to support the query optimizer in performing joins. Postgres can always evaluate a join (i.e., processing a clause with two tuple variables separated by an operator that returns a boolean) by iterative substitution [WONG76]. In addition, Postgres can use a hash-join algorithm along the lines of [SHAP86]; 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 hashjoined.) The HASHES flag indicates to the query optimizer that a hash join may safely be used with this operator.

Similarly, the two sort operators indicate to the query optimizer whether merge-sort is a usable join strategy and which operators should be used to sort the two operand classes. Sort operators should only be provided for an equality operator, and they should refer to less-than operators for the left and right side data types respectively.

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

The 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 Postgres may have to estimate the fraction of the instances in MYBOXES that satisfy the clause. The function *res_proc* must be a registered function (meaning it is already defined using **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 Postgres attempts to optimize operators and can decide to use an index to restrict the search space when operators are involved. However, there is no attempt to optimize functions, and they are performed by brute force. Moreover, functions can have any number of arguments while operators are restricted to one or two.

Notes

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

Usage

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

```
CREATE OPERATOR === (
   LEFTARG = box,
   RIGHTARG = box,
   PROCEDURE = area_equal_procedure,
   COMMUTATOR = ===,
   NEGATOR = !==,
   RESTRICT = area_restriction_procedure,
   JOIN = area_join_procedure,
   HASHES,
   SORT1 = <<<,
   SORT2 = <<<
);
```

Compatibility

SQL92

CREATE OPERATOR is a Postgres extension. There is no **CREATE OPERATOR** statement in SQL92.

CREATE RULE

Name

`CREATE RULE` Defines a new rule

Synopsis

```
CREATE RULE name AS ON event
    TO object [ WHERE condition ]
    DO [ INSTEAD ] action

where action can be:

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

Inputs

name

The name of a rule to create.

event

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

object

Object is either *table* or *table.column*. (Currently, only the *table* form is actually implemented.)

condition

Any SQL boolean-condition expression. The condition expression may not refer to any tables except `new` and `old`.

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 (the *object*). `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 SELECT, ON UPDATE, and ON DELETE rules to refer to the existing row being selected, updated, or deleted.

Outputs

`CREATE`

Message returned if the rule is successfully created.

Description

The Postgres *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 object (table) are examined, in an unspecified order. 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 before the original query is performed. 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 either parentheses or square brackets. Such queries will be performed in the specified order (whereas there are no guarantees about the execution order of multiple rules for an object). 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.

Notes

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 object 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 table and define an ON SELECT rule for it.

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 Postgres, the select command will cause Postgres to report an error because the query cycled too many times:

Example 1. Example of a circular rewrite rule combination:

```
CREATE RULE bad_rule_combination_1 AS
    ON SELECT TO emp
    DO INSTEAD
        SELECT * FROM toyemp;

CREATE RULE bad_rule_combination_2 AS
    ON SELECT TO toyemp
    DO INSTEAD
        SELECT * FROM emp;
```

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

```
SELECT * FROM emp;
```

Compatibility

SQL92

CREATE RULE statement is a Postgres language extension. There is no **CREATE RULE** statement in SQL92.

CREATE SEQUENCE

Name

`CREATE SEQUENCE` Creates a new sequence number generator

Synopsis

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

Inputs

seqname

The name of a sequence to be created.

increment

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

minvalue

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

maxvalue

The optional clause `MAXVALUE` *maxvalue* determines the maximum value for the sequence. The defaults are 2147483647 and -1 for ascending and descending sequences, respectively.

start

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

cache

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

CYCLE

The optional CYCLE keyword may be used to enable the sequence to 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.

Outputs

`CREATE`

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 data base. 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.

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

Use a query like

```
SELECT * FROM seqname;
```

to examine the parameters of a sequence. As an alternative to fetching the parameters from the original definition as above, you can use

```
SELECT last_value FROM seqname;
```

to obtain the last value allocated by any backend.

To avoid blocking of concurrent transactions that obtain numbers from the same sequence, a nextval operation is never rolled back; that is, once a value has been fetched it is considered used, even if the transaction that did the nextval later aborts. This means that aborted transactions may leave unused "holes" in the sequence of assigned values. setval operations are never rolled back, either.

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, numbers allocated but not used in the current session will be lost. Furthermore, although multiple backends are guaranteed to allocate distinct sequence values, the values may be generated out of sequence when all the backends are considered. (For example, with a cache setting of 10, backend A might reserve values 1..10 and return nextval=1, then backend B might reserve values 11..20 and return nextval=11 before backend A has generated nextval=2.) Thus, with a cache setting of one it is safe to assume that nextval values are generated sequentially; with a cache setting greater than one you should only assume that the nextval values are all distinct, not that they are generated purely sequentially. Also, last_value will reflect the latest value reserved by any backend, whether or not it has yet been returned by nextval. 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.

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');
```

Set the sequence value after a COPY FROM:

```
CREATE FUNCTION distributors_id_max() RETURNS INT4
    AS 'SELECT max(id) FROM distributors'
    LANGUAGE 'sql';
```

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

Compatibility

SQL92

CREATE SEQUENCE is a Postgres language extension. There is no **CREATE SEQUENCE** statement in SQL92.

CREATE TABLE

Name

`CREATE TABLE` Creates a new table

Synopsis

```
CREATE [ TEMPORARY | TEMP ] TABLE table_name (
    { column_name type [ column_constraint [ ... ] ]
      | table_constraint }  [, ... ]
    ) [ INHERITS ( inherited_table [, ... ] ) ]

where column_constraint can be:
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | UNIQUE | PRIMARY KEY | DEFAULT value | CHECK
(condition) |
  REFERENCES table [ ( column ) ] [ MATCH FULL | MATCH PARTIAL ]
   [ ON DELETE action ] [ ON UPDATE action ]
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY
IMMEDIATE ]
}

and table_constraint can be:
[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ... ] ) |
  PRIMARY KEY ( column_name [, ... ] ) |
  CHECK ( condition ) |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES table [ ( column [, ...
] ) ]
   [ MATCH FULL | MATCH PARTIAL ] [ ON DELETE action ] [ ON UPDATE action
]
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY
IMMEDIATE ]
}
```

Inputs

TEMPORARY or TEMP

If specified, the table is created only for this session, and is automatically dropped on session exit. Existing permanent tables with the same name are not visible (in this session) while the temporary table exists. Any indexes created on a temporary table are automatically temporary as well.

`table_name`

The name of the new table to be created.

`column_name`

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

`type`

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

`inherited_table`

The optional INHERITS clause specifies a list of table names from which this table automatically inherits all fields. If any inherited field name appears more than once, Postgres reports an error. Postgres automatically allows the created table to inherit functions on tables above it in the inheritance hierarchy.

`constraint_name`

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

`value`

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

`condition`

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.

`table`

The name of an existing table to be referenced by a foreign key constraint.

`column`

The name of a column in an existing table to be referenced by a foreign key constraint. If not specified, the primary key of the existing table is assumed.

`action`

A keyword indicating the action to take when a foreign key constraint is violated.

Outputs

`CREATE`

Message returned if table is successfully created.

`ERROR`

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

Description

CREATE TABLE will enter a new, initially empty table into the current database. The table will be "owned" by the user issuing the command.

Each *type* may be a simple type, a complex type (set) or an array type. Each attribute may be specified to be non-null and each may have a default value, specified by the *DEFAULT Clause*.

> **Note:** Consistent array dimensions within an attribute are not enforced. This will likely change in a future release.

CREATE TABLE also automatically creates a data type that represents the tuple type (structure type) corresponding to one row of the table. Therefore, tables can't have the same name as any existing datatype.

The optional INHERITS clause specifies a collection of table names from which this table automatically inherits all fields. If any inherited field name appears more than once, Postgres reports an error. Postgres automatically allows the created table to inherit functions on tables above it in the inheritance hierarchy. Inheritance of functions is done according to the conventions of the Common Lisp Object System (CLOS).

A table can have no more than 1600 columns (in practice, the effective limit is lower because of tuple-length constraints). A table cannot have the same name as a system catalog table.

DEFAULT Clause

```
DEFAULT value
```

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

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

Usage

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

The above 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.

It is worth remarking that

```
    modtime  TIMESTAMP DEFAULT 'now'
```

would produce a result that is probably not the intended one: the string `'now'` will be coerced to a timestamp value immediately, and so the default value of `modtime` will always be the time of table creation. This difficulty is avoided by specifying the default value as a function call.

Column Constraints

```
[ CONSTRAINT constraint_name ] {
    NULL | NOT NULL | UNIQUE | PRIMARY KEY | CHECK condition |
    REFERENCES reftable [ ( refcolumn ) ]
     [ MATCH matchtype ]
     [ ON DELETE action ]
     [ ON UPDATE action ]
     [ [ NOT ] DEFERRABLE ]
     [ INITIALLY checktime ] }
```

Inputs

constraint_name

An arbitrary name given to a constraint clause.

NULL

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

NOT NULL

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

UNIQUE

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

PRIMARY KEY

This column is a primary key, which implies that other tables may rely on this column as a unique identifier for rows. Both UNIQUE and NOT NULL are implied by PRIMARY KEY. See PRIMARY KEY for more information.

condition

An arbitrary boolean-valued constraint condition.

Description

The optional constraint clauses specify constraints or tests which 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 integrity constraints: table constraints, covered later, and column constraints, covered here.

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

PRIMARY KEY
REFERENCES
UNIQUE
CHECK
NOT NULL

NOT NULL Constraint

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

The NOT NULL constraint specifies a rule that a column may contain only non-null values. This is a column constraint only, and not allowed as a table constraint.

Outputs

status

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

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

Description

Usage

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

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

UNIQUE Constraint

`[ CONSTRAINT` *constraint_name* `] UNIQUE`

Inputs

constraint_name

An arbitrary name given to a constraint clause.

Outputs

status

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

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

Description

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

The column definitions of the specified columns do not have to include a NOT NULL constraint to be included in a UNIQUE constraint. Having more than one null value in a column without a NOT NULL constraint, does not violate a UNIQUE constraint. (This deviates from the SQL92 definition, but is a more sensible convention. See the section on compatibility for more details.)

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

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

Usage

Defines a UNIQUE constraint for the `name` column:

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

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

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

The CHECK Constraint

`[ CONSTRAINT` *`constraint_name`* `] CHECK (` *`condition`* `)`

Inputs

`constraint_name`

An arbitrary name given to a constraint clause.

`condition`

Any valid conditional expression evaluating to a boolean result.

Outputs

status

`ERROR: ExecAppend: rejected due to CHECK constraint "`*`constraint_name`*`".`

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

Description

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

CHECK specifies a general boolean expression involving one or more columns of a table. A new row will be rejected if the boolean expression evaluates to FALSE when applied to the row's values.

Currently, CHECK expressions cannot contain sub-selects nor refer to variables other than fields of the current row.

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. Postgres does not enforce this restriction. It treats column and table CHECK constraints alike.

PRIMARY KEY Constraint

```
[ CONSTRAINT constraint_name ] PRIMARY KEY
```

Inputs

constraint_name

An arbitrary name given to a constraint clause.

Outputs

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

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

Description

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

Only one PRIMARY KEY can be specified for a table, whether as a column constraint or a table constraint.

Notes

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

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

REFERENCES Constraint

```
[ CONSTRAINT constraint_name ] REFERENCES reftable [ ( refcolumn ) ]
    [ MATCH matchtype ]
    [ ON DELETE action ]
    [ ON UPDATE action ]
    [ [ NOT ] DEFERRABLE ]
    [ INITIALLY checktime ]
```

The REFERENCES constraint specifies a rule that a column value is checked against the values of another column. REFERENCES can also be specified as part of a FOREIGN KEY table constraint.

Inputs

constraint_name

An arbitrary name given to a constraint clause.

reftable

The table that contains the data to check against.

refcolumn

The column in *reftable* to check the data against. If this is not specified, the PRIMARY KEY of the *reftable* is used.

MATCH *matchtype*

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 multi-column 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 currently not supported.

ON DELETE *action*

The action to do when a referenced row in the referenced table is being deleted. There are the following actions.

NO ACTION

Produce error if foreign key violated. This is the default.

RESTRICT

Same as NO ACTION.

CASCADE

Delete any rows referencing the deleted row.

SET NULL

Set the referencing column values to NULL.

SET DEFAULT

Set the referencing column values to their default value.

ON UPDATE *`action`*

The action to do 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 changed, no action is done. There are the following actions.

NO ACTION

Produce error if foreign key violated. This is the default.

RESTRICT

Same as NO ACTION.

CASCADE

Update the value of the referencing column to the new value of the referenced column.

SET NULL

Set the referencing column values to NULL.

SET DEFAULT

Set the referencing column values to their default value.

[NOT] DEFERRABLE

This controls whether the constraint can be deferred to the end of the transaction. If DEFERRABLE, SET CONSTRAINTS ALL DEFERRED will cause the foreign key to be checked only at the end of the transaction. NOT DEFERRABLE is the default.

INITIALLY *`checktime`*

`checktime` has two possible values which specify the default time to check the constraint.

DEFERRED

Check constraint only at the end of the transaction.

IMMEDIATE

Check constraint after each statement. This is the default.

Outputs

status

`ERROR: name referential integrity violation - key referenced from table not found in reftable`

This error occurs at runtime if one tries to insert a value into a column which does not have a matching column in the referenced table.

Description

The REFERENCES column constraint specifies that a column of a table must only contain values which match against values in a referenced column of a referenced table.

A value added to this column is matched against the values of the referenced table and referenced column using the given match type. In addition, when the referenced column data is changed, actions are run upon this column's matching data.

Notes

Currently Postgres only supports MATCH FULL and a default match type. In addition, the referenced columns are supposed to be the columns of a UNIQUE constraint in the referenced table, however Postgres does not enforce this.

Table Constraints

```
[ CONSTRAINT name ] { PRIMARY KEY |  UNIQUE } ( column [, ... ] )
[ CONSTRAINT name ] CHECK ( constraint )
[ CONSTRAINT name ] FOREIGN KEY ( column [, ... ] )
                     REFERENCES reftable [ ( refcolumn [, ... ] ) ]
                      [ MATCH matchtype ]
                      [ ON DELETE action ]
                      [ ON UPDATE action ]
                      [ [ NOT ] DEFERRABLE ]
                      [ INITIALLY checktime ]
```

Inputs

constraint_name

An arbitrary name given to a constraint clause.

column [, ...]

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

CHECK (*constraint*)

A boolean expression to be evaluated as the constraint.

Outputs

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

Description

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

UNIQUE
CHECK
PRIMARY KEY
FOREIGN KEY

UNIQUE Constraint

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

Inputs

constraint_name

An arbitrary name given to a constraint clause.

column

A name of a column in a table.

Outputs

status

ERROR: Cannot insert a duplicate key into a unique index

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

Description

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

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

Usage

Prevent duplicate rows in the table distributors:

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

PRIMARY KEY Constraint

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

Inputs

constraint_name

An arbitrary name given to a constraint clause.

column [, ...]

The names of one or more columns in the table.

Outputs

status

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

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

Description

The PRIMARY KEY constraint specifies a rule that a group of one or more distinct columns of a table may contain only unique (nonduplicate), non-null values. The column definitions of the specified columns do not have to include a NOT NULL constraint to be included in a PRIMARY KEY constraint.

The PRIMARY KEY table constraint is similar to that for column constraints, with the additional capability of encompassing multiple columns.

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

REFERENCES Constraint

```
[ CONSTRAINT constraint_name ] FOREIGN KEY ( column [, ... ] )
    REFERENCES reftable [ ( refcolumn [, ... ] ) ]
    [ MATCH matchtype ]
    [ ON DELETE action ]
    [ ON UPDATE action ]
    [ [ NOT ] DEFERRABLE ]
    [ INITIALLY checktime ]
```

The REFERENCES constraint specifies a rule that a column value or set of column values is checked against the values in another table.

Inputs

constraint_name

An arbitrary name given to a constraint clause.

column [, ...]

The names of one or more columns in the table.

reftable

The table that contains the data to check against.

referenced column [, ...]

One or more columns in the *reftable* to check the data against. If this is not specified, the PRIMARY KEY of the *reftable* is used.

MATCH *matchtype*

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 multi-column 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 currently not supported.

ON DELETE *action*

The action to do when a referenced row in the referenced table is being deleted. There are the following actions.

NO ACTION

Produce error if foreign key violated. This is the default.

RESTRICT

Same as NO ACTION.

CASCADE

Delete any rows referencing the deleted row.

SET NULL

Set the referencing column values to NULL.

SET DEFAULT

Set the referencing column values to their default value.

ON UPDATE *`action`*

The action to do 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 changed, no action is done. There are the following actions.

NO ACTION

Produce error if foreign key violated. This is the default.

RESTRICT

Disallow update of row being referenced.

CASCADE

Update the value of the referencing column to the new value of the referenced column.

SET NULL

Set the referencing column values to NULL.

SET DEFAULT

Set the referencing column values to their default value.

[NOT] DEFERRABLE

This controls whether the constraint can be deferred to the end of the transaction. If DEFERRABLE, SET CONSTRAINTS ALL DEFERRED will cause the foreign key to be checked only at the end of the transaction. NOT DEFERRABLE is the default.

INITIALLY *`checktime`*

`checktime` has two possible values which specify the default time to check the constraint.

IMMEDIATE

Check constraint after each statement. This is the default.

DEFERRED

Check constraint only at the end of the transaction.

Outputs

status

`ERROR: name referential integrity violation - key referenced from table not found in reftable`

This error occurs at runtime if one tries to insert a value into a column which does not have a matching column in the referenced table.

Description

The FOREIGN KEY constraint specifies a rule that a group of one or more distinct columns of a table is related to a group of distinct columns in the referenced table.

The FOREIGN KEY table constraint is similar to that for column constraints, with the additional capability of encompassing multiple columns.

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

Usage

Create table films and table distributors:

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

```
CREATE TABLE distributors (
     did      DECIMAL(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      CHAR(10),
    len       INTERVAL HOUR TO MINUTE,
    CONSTRAINT production UNIQUE(date_prod)
);
```

Define a CHECK column constraint:

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

Define a CHECK table constraint:

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

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

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

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

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

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

Compatibility

SQL92

In addition to the locally visible temporary table, SQL92 also defines a CREATE GLOBAL TEMPORARY TABLE statement, and optionally an ON COMMIT clause:

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

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

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

UNIQUE clause

SQL92 specifies some additional capabilities for UNIQUE:

Table Constraint definition:

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

Column Constraint definition:

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

NULL clause

The NULL "constraint" (actually a non-constraint) is a Postgres extension to SQL92 that is included for symmetry with the NOT NULL clause (and for compatibility with some other RDBMSes). Since it is the default for any column, its presence is simply noise.

```
[ CONSTRAINT constraint_name ] NULL
```

NOT NULL clause

SQL92 specifies some additional capabilities for NOT NULL:

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

CONSTRAINT clause

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

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

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

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

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

Domain constraint:

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

Table constraint definition:

```
[ CONSTRAINT constraint_name ] { PRIMARY KEY ( column, ... ) | FOREIGN KEY
constraint | UNIQUE constraint | CHECK constraint }
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

Column constraint definition:

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

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

NOT DEFERRABLE

The constraint must be checked at the end of each statement. SET CONSTRAINTS ALL DEFERRED will have no effect on this type of constraint.

DEFERRABLE

This controls whether the constraint can be deferred to the end of the transaction. If SET CONSTRAINTS ALL DEFERRED is used or the constraint is set to INITIALLY DEFERRED, this will cause the foreign key to be checked only at the end of the transaction.

> **Note: SET CONSTRAINTS** changes the foreign key constraint mode only for the current transaction.

INITIALLY IMMEDIATE

Check constraint after each statement. This is the default.

INITIALLY DEFERRED

Check constraint only at the end of the transaction.

CHECK clause

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

table constraint definition:

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

column constraint definition:

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

PRIMARY KEY clause

SQL92 specifies some additional capabilities for PRIMARY KEY:

Table Constraint definition:

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

Column Constraint definition:

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

Inheritance

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

CREATE TABLE AS

Name

`CREATE TABLE AS` Creates a new table from the results of a SELECT

Synopsis

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

Inputs

TEMPORARY or TEMP

If specified, the table is created only within this session, and is automatically dropped on session exit. Existing permanent tables with the same name are not visible (in this session) while the temporary table exists. Any indexes created on a temporary table are automatically temporary as well.

table

The name of the new table to be created. This table must not already exist. However, a temporary table can be created that has the same name as an existing permanent table.

column

The name of a column. 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 SELECT query.

select_clause

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

Outputs

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

Description

CREATE TABLE AS creates a table and fills it with data computed by a **SELECT** command. The table columns have the names and datatypes associated with the output columns of the **SELECT** (except that you can override the **SELECT** column names by giving an explicit list of 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 **SELECT** just once to fill the new table initially. The new table will not track subsequent changes to the source tables of the **SELECT**. In contrast, a view re-evaluates the given **SELECT** whenever queried.

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.

CREATE TRIGGER

Name

`CREATE TRIGGER` Creates 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 of an existing trigger.

table

The name of a table.

event

One of INSERT, DELETE or UPDATE.

func

A user-supplied function.

Outputs

CREATE

This message is returned if the trigger is successfully created.

Description

CREATE TRIGGER will enter a new trigger into the current data base. The trigger will be associated with the relation *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.

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

Notes

CREATE TRIGGER is a Postgres language extension.

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

As of the current release (v7.0), STATEMENT triggers are not implemented.

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

Usage

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

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

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

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

Compatibility

SQL92

There is no **CREATE TRIGGER** in SQL92.

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

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

CREATE TYPE

Name

CREATE TYPE   Defines a new base data type

Synopsis

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

Inputs

typename

The name of a type to be created.

internallength

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

externallength

A literal value, which specifies the external (displayed) 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 for the array elements.

default

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

send_function

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

receive_function

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

alignment

Storage alignment requirement of the data type. If specified, must be '`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`'.

Outputs

`CREATE`

Message returned if the type is successfully created.

Description

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

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

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

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

```
ELEMENT = int4
```

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

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

* *How does the user specify that bit pattern and associate it with the fact that the data is not present>*

The optional arguments *`send_function`* and *`receive_function`* are used when the application program requesting Postgres services resides on a different machine. In this case, the machine on which Postgres runs may use a format for the data type different from that used on the remote machine. In this case it is appropriate to convert data items to a standard form when sending from the server to the client and converting from the standard format to the machine specific format when the server receives the data from the client. If these functions are not specified, then it is assumed that the internal format of the type is acceptable on all relevant machine architectures. For example, single characters do not have to be converted if passed from a Sun-4 to a DECstation, but many other types do.

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

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.)

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

Array Types

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

Examples

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

```
CREATE TYPE box (INTERNALLENGTH = 8,
    INPUT = my_procedure_1, OUTPUT = my_procedure_2);
CREATE TABLE myboxes (id INT4, description box);
```

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

```
CREATE TYPE int4array (INPUT = array_in, OUTPUT = array_out,
    INTERNALLENGTH = VARIABLE, ELEMENT = int4);
CREATE TABLE myarrays (id int4, numbers int4array);
```

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

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

Notes

Type names cannot begin with the underscore character ("_") and can only be 31 characters long. This is because Postgres silently creates an array type for each base type with a name consisting of the base type's name prepended with an underscore.

Refer to **DROP TYPE** to remove an existing type.

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

Compatibility

SQL3

CREATE TYPE is an SQL3 statement.

CREATE USER

Name

`CREATE USER` Creates a new database user

Synopsis

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

Inputs

username

The name of the user.

uid

The `SYSID` clause can be used to choose the Postgres 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 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, otherwise the user won't be able to connect to a password-authenticated server. See the chapter on client authentication in the *Administrator's Guide* for details on how to set up authentication mechanisms.

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.

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.

Outputs

`CREATE USER`

Message returned if the command completes successfully.

Description

CREATE USER will add a new user to an instance of Postgres. Refer to the administrator's guide for information about managing users and authentication. You must be a database superuser to use this command.

Use *ALTER USER* to change a user's password and privileges, and *DROP USER* to remove a user. Use **ALTER GROUP** to add or remove the user from other groups. Postgres comes with a script *createuser* which has the same functionality as this command (in fact, it calls this command) but can be run from the command shell.

Usage

Create a user with no password:

```
CREATE USER jonathan
```

Create a user with a password:

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

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

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

Create an account where the user can create databases:

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

Compatibility

SQL92

There is no **CREATE USER** statement in SQL92.

CREATE VIEW

Name

`CREATE VIEW` Constructs a virtual table

Synopsis

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

Inputs

view

The name of a view to be created.

query

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

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

Outputs

`CREATE`

The message returned if the view is successfully created.

`ERROR: Relation 'view' already exists`

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

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

The view will be created having a column with an unknown type if you do not specify it. For example, the following command gives 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 will define a view of a table. This view is not physically materialized. Specifically, a query rewrite retrieve rule is automatically generated to support retrieve operations on views.

Notes

Currently, views are read only.

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 INSERTs and UPDATEs on the view will be checked to ensure data satisfy the view-defining condition. If they do not, the update will be rejected.

LOCAL

: Check for integrity on this view.

CASCADE

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

DECLARE

Name

`DECLARE` Defines a cursor for table access

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 Postgres this is always the case. This keyword has no effect.

SCROLL

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

query

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

READ ONLY

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

UPDATE

SQL92 keyword indicating that the cursor will be used to update tables. Since cursor updates are not currently supported in Postgres this keyword provokes an informational error message.

column

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

Outputs

`SELECT`

The message returned if the SELECT is run successfully.

`NOTICE: 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 Postgres backend was built. Since data is stored natively in binary format, the system must do a conversion to produce the text format. In addition, text formats are often larger in size than the corresponding binary format. Once the information comes back in text form, the client application may 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. *Postgres 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 Postgres 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 Postgres, 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. Postgres permits cursors to be used interactively. SQL92 allows embedded or modular cursors to update database information. All Postgres cursors are read only. The BINARY keyword is a Postgres extension.

DELETE

Name

`DELETE` Removes rows from a table

Synopsis

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

Inputs

`table`

The name of an existing table.

`condition`

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

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

Outputs

`DELETE count`

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

If *count* is 0, no rows were deleted.

Description

DELETE removes rows which satisfy the WHERE 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 Postgres 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 Postgres are read-only.

DROP AGGREGATE

Name

DROP AGGREGATE Removes the definition of an aggregate function

Synopsis

```
DROP AGGREGATE name type
```

Inputs

name

The name of an existing aggregate function.

type

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

* *This should become a cross-reference rather than a hard-coded chapter number*

Outputs

DROP

Message returned if the command is successful.

ERROR: RemoveAggregate: aggregate '*agg*' for '*type*' does not exist

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

Description

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

Notes

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 Postgres language extension.

DROP DATABASE

Name

`DROP DATABASE` Removes an existing 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.

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

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

`ERROR: dropdb: cannot be executed on the template database`

The `template1` database cannot be removed. It's not in your interest.

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

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

`ERROR: dropdb: database 'name' does not exist`

This message occurs if the specified database does not exist.

`ERROR: dropdb: database 'name' is not owned by you`

You must be the owner of the database. Being the owner usually means that you created it as well.

`ERROR: dropdb: May not be called in a transaction block.`

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

`NOTICE: The database directory 'xxx' could not be removed.`

The database was dropped (unless other error messages came up), but the directory where the data is stored could not be removed. You must delete it manually.

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).

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 Postgres language extension; there is no such command in SQL92.

DROP FUNCTION

Name

`DROP FUNCTION` Removes a user-defined C function

Synopsis

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

Inputs

name

The name of an existing function.

type

The type of function parameters.

Outputs

`DROP`

Message returned if the command completes successfully.

`NOTICE RemoveFunction: Function "name" ("types") does not exist`

This message is given if the function specified does not exist in the current database.

Description

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

Notes

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

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

Usage

This command removes the square root function:

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

Compatibility

SQL92

DROP FUNCTION is a Postgres language extension.

SQL/PSM

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

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

DROP GROUP

Name

`DROP GROUP` Removes a 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` Removes existing indexes from a database

Synopsis

```
DROP INDEX index_name [, ...]
```

Inputs

`index_name`

The name of an index to remove.

Outputs

`DROP`

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 Postgres 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` Removes a user-defined procedural language

Synopsis

```
DROP [ PROCEDURAL ] LANGUAGE 'name'
```

Inputs

name

The name of an existing procedural language.

Outputs

`DROP`

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 Postgres language extension.

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

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

Usage

This command removes the PL/Sample language:

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

Compatibility

SQL92

There is no **DROP PROCEDURAL LANGUAGE** in SQL92.

DROP OPERATOR

Name

`DROP OPERATOR` Removes an operator from the database

Synopsis

```
DROP OPERATOR id ( lefttype | NONE , righttype | NONE )
```

Inputs

id

The identifier 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.

Outputs

`DROP`

The message returned if the command is successful.

`ERROR: RemoveOperator: binary operator 'oper' taking 'type' and 'type2' does not exist`

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

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

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

`ERROR: RemoveOperator: right unary operator 'oper' taking 'type' 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 Postgres language extension.

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

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

Usage

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

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

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

```
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 RULE

Name

`DROP RULE` Removes existing rules from the database

Synopsis

```
DROP RULE name [, ...]
```

Inputs

name

The name of an existing rule to drop.

Outputs

`DROP`

Message returned if successful.

`ERROR: Rule or view "name" not found`

This message occurs if the specified rule does not exist.

Description

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

Notes

The **DROP RULE** statement is a Postgres language extension.

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

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

Usage

To drop the rewrite rule `newrule`:

```
DROP RULE newrule;
```

Compatibility

SQL92

There is no **DROP RULE** in SQL92.

DROP SEQUENCE

Name

`DROP SEQUENCE` Removes existing sequences from a database

Synopsis

```
DROP SEQUENCE name [, ...]
```

Inputs

`name`

The name of a sequence.

Outputs

`DROP`

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 Postgres language extension.

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

Usage

To remove sequence `serial` from database:

```
DROP SEQUENCE serial;
```

Compatibility

SQL92

There is no **DROP SEQUENCE** in SQL92.

DROP TABLE

Name

`DROP TABLE` Removes existing tables from a database

Synopsis

```
DROP TABLE name [, ...]
```

Inputs

`name`

The name of an existing table to drop.

Outputs

`DROP`

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**.

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

Notes

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

Usage

To destroy two tables, `films` and `distributors`:

```
DROP TABLE films, distributors;
```

Compatibility

SQL92

SQL92 specifies some additional capabilities for DROP TABLE:

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

RESTRICT

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

CASCADE

Any referencing views or integrity constraints will also be dropped.

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

DROP TRIGGER

Name

`DROP TRIGGER` Removes the definition of a trigger

Synopsis

```
DROP TRIGGER name ON table
```

Inputs

`name`

The name of an existing trigger.

`table`

The name of a table.

Outputs

`DROP`

The message returned if the trigger is successfully dropped.

`ERROR: DropTrigger: there is no trigger name on relation "table"`

This message occurs if the trigger specified does not exist.

Description

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

Notes

DROP TRIGGER is a Postgres language extension.

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

Usage

Destroy the `if_dist_exists` trigger on table `films`:

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

Compatibility

SQL92

There is no **DROP TRIGGER** statement in SQL92.

DROP TYPE

Name

`DROP TYPE` Removes user-defined types from the system catalogs

Synopsis

```
DROP TYPE typename [, ...]
```

Inputs

typename

The name of an existing type.

Outputs

`DROP`

The message returned if the command is successful.

`ERROR: RemoveType: type 'typename' does not exist`

This message occurs if the specified type is not found.

Description

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

Only the owner of a type can remove it.

Notes

DROP TYPE statement is a Postgres language extension.

Refer to **CREATE TYPE** for information on how to create types.

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

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

Usage

To remove the `box` type:

```
DROP TYPE box;
```

Compatibility

SQL3

DROP TYPE is a SQL3 statement.

DROP USER

Name

`DROP USER` Removes a user

Synopsis

```
DROP USER name
```

Inputs

name

The name of an existing user.

Outputs

`DROP USER`

The message returned if the user is successfully deleted.

`ERROR: DROP USER: user "name" does not exist`

This message occurs if the username is not found.

`DROP USER: user "name" owns database "name", cannot be removed`

You must drop the database first or change its ownership.

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 you get an error.

Use *CREATE USER* to add new users, and *ALTER USER* to change a user's properties. Postgres comes with a script *dropuser* which has the same functionality as this command (in fact, it calls this command) but can be run from the command shell.

Usage

To drop a user account:

```
DROP USER jonathan;
```

Compatibility

SQL92

There is no **DROP USER** in SQL92.

DROP VIEW

Name

`DROP VIEW` Removes existing views from a database

Synopsis

```
DROP VIEW name [, ...]
```

Inputs

name

The name of an existing view.

Outputs

`DROP`

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

SQL92 specifies some additional capabilities for **DROP VIEW**:

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

Inputs

RESTRICT

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

CASCADE

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

Notes

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

END

Name

`END` Commits 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.

`NOTICE: COMMIT: no transaction in progress`

If there is no transaction in progress.

Description

END is a Postgres 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*.

EXPLAIN

Name

`EXPLAIN` Shows statement execution plan

Synopsis

```
EXPLAIN [ VERBOSE ] query
```

Inputs

VERBOSE

Flag to show detailed query plan.

query

Any *query*.

Outputs

`NOTICE: QUERY PLAN:` *plan*

Explicit query plan from the Postgres backend.

`EXPLAIN`

Flag sent after query plan is shown.

Description

This command displays the execution plan that the Postgres 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 sub-query 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 VERBOSE option emits the full internal representation of the plan tree, rather than just a summary (and sends it to the postmaster log file, too). Usually this option is only useful for debugging Postgres.

Notes

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

Usage

To show a query plan for a simple query on a table with a single `int4` column and 128 rows:

```
EXPLAIN SELECT * FROM foo;
    NOTICE:  QUERY PLAN:

Seq Scan on foo  (cost=0.00..2.28 rows=128 width=4)

EXPLAIN
```

For the same table with an index to support an *equijoin* condition on the query, **EXPLAIN** will show a different plan:

```
EXPLAIN SELECT * FROM foo WHERE i = 4;
    NOTICE:  QUERY PLAN:

Index Scan using fi on foo  (cost=0.00..0.42 rows=1 width=4)

EXPLAIN
```

And finally, for the same table with an index to support an *equijoin* condition on the query, **EXPLAIN** will show the following for a query using an aggregate function:

```
EXPLAIN SELECT sum(i) FROM foo WHERE i = 4;
    NOTICE:  QUERY PLAN:

Aggregate  (cost=0.42..0.42 rows=1 width=4)
  ->  Index Scan using fi on foo  (cost=0.00..0.42 rows=1 width=4)
```

Note that the specific numbers shown, and even the selected query strategy, may vary between Postgres releases due to planner improvements.

Compatibility

SQL92

There is no **EXPLAIN** statement defined in SQL92.

FETCH

Name

`FETCH` Gets rows 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:

`NOTICE: PerformPortalFetch: portal "cursor" not found`

If *cursor* is not previously declared. The cursor must be declared within a transaction block.

`NOTICE: FETCH/ABSOLUTE not supported, using RELATIVE`

Postgres does not support absolute positioning of cursors.

`ERROR: FETCH/RELATIVE at current position is not supported`

SQL92 allows one to repetitively retrieve the cursor at its "current position" using the syntax

```
FETCH RELATIVE 0 FROM cursor.
```

Postgres does not currently support this notion; in fact the value zero is reserved to indicate that all rows should be retrieved and is equivalent to specifying the ALL keyword. If the RELATIVE keyword has been used, Postgres assumes that the user intended SQL92 behavior and returns this error message.

Description

FETCH allows a user to retrieve rows using a cursor. The number of rows retrieved is specified by #. If the number of rows remaining in the cursor is less than #, then only those available are fetched. Substituting the keyword ALL in place of a number will cause all remaining rows in the cursor to be retrieved. Instances may be fetched in both FORWARD and BACKWARD directions. The default direction is FORWARD.

Tip: Negative numbers are 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 Postgres 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 Postgres, because mapping cursor updates back to base tables is not generally possible, as is also the case with VIEW updates. Consequently, users must issue explicit UPDATE commands to replace data.

Cursors may only be used inside of transactions because the data that they store spans multiple user queries.

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 Postgres 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 Postgres are relative numbers so this capability is not supported.

:*variable*

Target host variable(s).

GRANT

Name

`GRANT` Grants access privilege to a user, a group or all users

Synopsis

```
GRANT privilege [, ...] ON object [, ...]
    TO { PUBLIC | GROUP group | username }
```

Inputs

privilege

The possible privileges are:

SELECT

Access all of the columns of a specific table/view.

INSERT

Insert data into all columns of a specific table.

UPDATE

Update all columns of a specific table.

DELETE

Delete rows from a specific table.

RULE

Define rules on the table/view (See CREATE RULE statement).

ALL

Grant all privileges.

`object`

The name of an object to which to grant access. The possible objects are:
table
view
sequence

PUBLIC

A short form representing all users.

GROUP *`group`*

A *`group`* to whom to grant privileges.

`username`

The name of a user to whom to grant privileges. PUBLIC is a short form representing all users.

Outputs

`CHANGE`

Message returned if successful.

`ERROR: ChangeAcl: class "object" not found`

Message returned if the specified object is not available or if it is impossible to give privileges to the specified group or users.

Description

GRANT allows the creator of an object to give specific permissions to all users (PUBLIC) or to a certain user or group. Users other than the creator don't have any access permission unless the creator GRANTs permissions, after the object is created.

Once a user has a privilege on an object, he is enabled to exercise that privilege. There is no need to GRANT privileges to the creator of an object, the creator automatically holds ALL privileges, and can also drop the object.

Notes

Currently, to grant privileges in Postgres to only a few columns, you must create a view having desired columns and then grant privileges to that view.

Use **psql \z** for further information about permissions on existing objects:

```
      Database    = lusitania
+------------------+---------------------------------------------+
|  Relation        |        Grant/Revoke Permissions             |
+------------------+---------------------------------------------+
| mytable          | {"=rw","miriam=arwR","group todos=rw"}      |
+------------------+---------------------------------------------+
Legend:
      uname=arwR -- privileges granted to a user
group gname=arwR -- privileges granted to a GROUP
           =arwR -- privileges granted to PUBLIC

               r -- SELECT
               w -- UPDATE/DELETE
               a -- INSERT
               R -- RULE
            arwR -- ALL
```

Refer to REVOKE statements to revoke access privileges.

Usage

Grant insert privilege to all users on table films:

```
GRANT INSERT ON films TO PUBLIC;
```

Grant all privileges to user manuel on view kinds:

```
GRANT ALL ON kinds TO manuel;
```

Compatibility

SQL92

The SQL92 syntax for GRANT allows setting privileges for individual columns within a table, and allows setting a privilege to grant the same privileges to others:

```
GRANT privilege [, ...]
    ON object [ ( column [, ...] ) ] [, ...]
    TO { PUBLIC | username [, ...] } [ WITH GRANT OPTION ]
```

Fields are compatible with those in the Postgres implementation, with the following additions:

privilege

SQL92 permits additional privileges to be specified:

SELECT

REFERENCES

Allowed to reference some or all of the columns of a specific table/view in integrity constraints.

USAGE

Allowed to use a domain, character set, collation or translation. If an object specifies anything other than a table/view, *privilege* must specify only USAGE.

object

[TABLE] *table*

SQL92 allows the additional non-functional keyword `TABLE`.

CHARACTER SET

Allowed to use the specified character set.

COLLATION

Allowed to use the specified collation sequence.

TRANSLATION

Allowed to use the specified character set translation.

DOMAIN

Allowed to use the specified domain.

WITH GRANT OPTION

Allowed to grant the same privilege to others.

INSERT

Name

`INSERT` Inserts new rows into a table

Synopsis

```
INSERT INTO table [ ( column [, ...] ) ]
    { DEFAULT VALUES | VALUES ( expression [, ...] ) | SELECT query }
```

Inputs

table

The name of an existing table.

column

The name of a column in *table*.

DEFAULT VALUES

All columns will be filled by NULLs or by values specified when the table was created using DEFAULT clauses.

expression

A valid expression or value to assign to *column*.

query

A valid query. Refer to the SELECT statement for a further description of valid arguments.

Outputs

`INSERT oid 1`

Message returned if only one row was inserted. *oid* is the 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. Postgres 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');
```

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 response on a notify condition

Synopsis

```
LISTEN name
```

Inputs

name

Name of notify condition.

Outputs

`LISTEN`

Message returned upon successful completion of registration.

`NOTICE Async_Listen: We are already listening on name`

If this backend is already registered for that notify condition.

Description

LISTEN registers the current Postgres 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 Postgres application programming interface it uses. With the basic libpq library, the application issues **LISTEN** as an ordinary SQL command, and then must periodically call the routine `PQnotifies` to find out whether any notify events have been received. Other interfaces such as libpgtcl provide higher-level methods for handling notify events; indeed, with libpgtcl the application programmer should not even issue **LISTEN** or **UNLISTEN** directly. See the documentation for the library you are using for more details.

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 31 characters long.

In some previous releases of Postgres, *`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` Dynamically loads an object file

Synopsis

```
LOAD 'filename'
```

Inputs

`filename`

Object file for dynamic loading.

Outputs

`LOAD`

Message returned on successful completion.

`ERROR: LOAD: could not open file 'filename'`

Message returned if the specified file is not found. The file must be visible *to the Postgres backend*, with the appropriate full path name specified, to avoid this message.

Description

Loads an object (or ".o") file into the Postgres backend address space. Once a file is loaded, all functions in that file can be accessed. This function is used in support of user-defined types and functions.

If a file is not loaded using **LOAD**, the file will be loaded automatically the first time the function is called by Postgres. **LOAD** can also be used to reload an object file if it has been edited and recompiled. Only objects created from C language files are supported at this time.

Notes

Functions in loaded object files should not call functions in other object files loaded through the **LOAD** command. For example, all functions in file `A` should call each other, functions in the standard or math libraries, or in Postgres itself. They should not call functions defined in a different loaded file `B`. This is because if `B` is reloaded, the Postgres loader is not able to relocate the calls from the functions in `A` into the new address space of `B`. If `B` is not reloaded, however, there will not be a problem.

Object files must be compiled to contain position independent code. For example, on DECstations you must use /bin/cc with the `-G 0` option when compiling object files to be loaded.

Note that if you are porting Postgres to a new platform, **LOAD** will have to work in order to support ADTs.

Usage

Load the file `/usr/postgres/demo/circle.o`:

```
LOAD '/usr/postgres/demo/circle.o'
```

Compatibility

SQL92

There is no **LOAD** in SQL92.

LOCK

Name

`LOCK` Explicitly lock a table inside a transaction

Synopsis

```
LOCK [ TABLE ] name
LOCK [ TABLE ] name IN [ ROW | ACCESS ] { SHARE | EXCLUSIVE } MODE
LOCK [ TABLE ] name IN SHARE ROW EXCLUSIVE MODE
```

Inputs

name

The name of an existing table to lock.

ACCESS SHARE MODE

Note: This lock mode is acquired automatically over tables being queried.

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** commands.

ROW SHARE MODE

Note: Automatically acquired by **SELECT...FOR UPDATE**. While it is a shared lock, may be upgraded later to a ROW EXCLUSIVE lock.

Conflicts with EXCLUSIVE and ACCESS EXCLUSIVE lock modes.

ROW EXCLUSIVE MODE

Note: Automatically acquired by **UPDATE**, **DELETE**, and **INSERT** statements.

Conflicts with SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes.

SHARE MODE

Note: Automatically acquired by **CREATE INDEX**. Share-locks the entire table.

Conflicts with ROW EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes. This mode protects a table against concurrent updates.

SHARE ROW EXCLUSIVE MODE

> **Note:** This is like EXCLUSIVE MODE, but allows SHARE ROW locks by others.

Conflicts with ROW EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes.

EXCLUSIVE MODE

> **Note:** This mode is yet more restrictive than SHARE ROW EXCLUSIVE. It blocks all concurrent ROW SHARE/SELECT...FOR UPDATE queries.

Conflicts with ROW SHARE, ROW EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes.

ACCESS EXCLUSIVE MODE

> **Note:** Automatically acquired by **ALTER TABLE**, **DROP TABLE**, **VACUUM** statements. This is the most restrictive lock mode which conflicts with all other lock modes and protects a locked table from any concurrent operations.
>
> **Note:** This lock mode is also acquired by an unqualified **LOCK TABLE** (i.e., the command without an explicit lock mode option).

Outputs

`LOCK TABLE`

The lock was successfully applied.

`ERROR` *`name`*`: Table does not exist.`

Message returned if *`name`* does not exist.

Description

LOCK TABLE controls concurrent access to a table for the duration of a transaction. Postgres always uses the least restrictive lock mode whenever possible. **LOCK TABLE** provides for cases when you might need more restrictive locking.

RDBMS locking uses the following terminology:

EXCLUSIVE

Exclusive lock that prevents other locks from being granted.

SHARE

Allows others to share lock. Prevents EXCLUSIVE locks.

ACCESS

Locks table schema.

ROW

Locks individual rows.

> **Note:** If EXCLUSIVE or SHARE are not specified, EXCLUSIVE is assumed. Locks exist for the duration of the transaction.

For example, an application runs a transaction at READ COMMITTED isolation level and needs to ensure the existence of data in a table for the duration of the transaction. To achieve this you could use SHARE lock mode over the table before querying. This will protect data from concurrent changes and provide any further read operations over the table with data in their actual current state, because SHARE lock mode conflicts with any ROW EXCLUSIVE one acquired by writers, and your **LOCK TABLE *name* IN SHARE MODE** statement will wait until any concurrent write operations commit or rollback.

> **Note:** To read data in their real current state when running a transaction at the SERIALIZABLE isolation level you have to execute a LOCK TABLE statement before executing any DML statement, when the transaction defines what concurrent changes will be visible to itself.

In addition to the requirements above, if a transaction is going to change data in a table, then SHARE ROW EXCLUSIVE lock mode should be acquired to prevent deadlock conditions when two concurrent transactions attempt to lock the table in SHARE mode and then try to change data in this table, both (implicitly) acquiring ROW EXCLUSIVE lock mode that conflicts with a concurrent SHARE lock.

To continue with the deadlock (when two transaction wait for one another) issue raised above, you should follow two general rules 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.

Transactions should acquire two conflicting lock modes only if one of them is self-conflicting (i.e., may be held by one transaction at time only). If multiple lock modes are involved, 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.

> **Note:** Postgres does detect deadlocks and will rollback at least one waiting transaction to resolve the deadlock.

Notes

LOCK is a Postgres language extension.

Except for ACCESS SHARE/EXCLUSIVE lock modes, all other Postgres lock modes and the **LOCK TABLE** syntax are compatible with those present in Oracle.

LOCK works only inside transactions.

Usage

Illustrate 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.

MOVE

Name

MOVE — Moves cursor position

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 Postgres 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` Signals all frontends and backends listening on a notify condition

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 Postgres database. Higher-level mechanisms can be built by using tables in the database to pass additional data (beyond a mere condition name) from notifier to listener(s).

When **NOTIFY** is used to signal the occurrence of changes to a particular table, a useful programming technique is to put the **NOTIFY** in a rule that is triggered by table updates. In this way, notification happens automatically when the table is changed, and the application programmer can't accidentally forget to do it.

NOTIFY interacts with SQL transactions in some important ways. Firstly, if a **NOTIFY** is executed inside a transaction, the notify events are not delivered until and unless the transaction is committed. This is appropriate, since if the transaction is aborted we would like all the commands within it to have had no effect, including **NOTIFY**. But it can be disconcerting if one is expecting the notify events to be delivered immediately. Secondly, if a listening backend receives a notify signal while it is within a transaction, the notify event will not be delivered to its connected frontend until just after the transaction is completed (either committed or aborted). Again, the reasoning is that if a notify were delivered within a transaction that was later aborted, one would want the notification to be undone somehow---but the backend cannot "take back" a notify once it has sent it to the frontend. So notify events are only delivered between transactions. The upshot of this is that applications using **NOTIFY** for real-time signaling should try to keep their transactions short.

NOTIFY behaves like Unix signals in one important respect: if the same condition name is signaled multiple times in quick succession, recipients may get only one notify event for several executions of **NOTIFY**. So it is a bad idea to depend on the number of notifies received. Instead, use **NOTIFY** to wake up applications that need to pay attention to something, and use a database object (such as a sequence) to keep track of what happened or how many times it happened.

It is common for a frontend that sends **NOTIFY** to be listening on the same notify name itself. In that case it will get back a notify event, just like all the other listening frontends. Depending on the application logic, this could result in useless work---for example, re-reading a database table to find the same updates that that frontend just wrote out. In Postgres 6.4 and later, it is possible to avoid such extra work by noticing whether the notifying backend process's PID (supplied in the notify event message) is the same as one's own backend's PID (available from libpq). When they are the same, the notify event is one's own work bouncing back, and can be ignored. (Despite what was said in the preceding paragraph, this is a safe technique. Postgres keeps self-notifies separate from notifies arriving from other backends, so you cannot miss an outside notify by ignoring your own notifies.)

Notes

`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 31 characters long.

In some previous releases of Postgres, `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 Postgres releases prior to 6.4, the backend PID delivered in a notify message was always the PID of the frontend's own backend. So it was not possible to distinguish one's own notifies from other clients' notifies in those earlier releases.

Usage

Configure and execute a listen/notify sequence from psql:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous NOTIFY 'virtual' from backend with pid '8448' received.
```

Compatibility

SQL92

There is no **NOTIFY** statement in SQL92.

REINDEX

Name

`REINDEX` Recover corrupted system indexes under stand-alone Postgres

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.

INDEX

Recreate a specified index.

name

The name of the specific table/database/index to be be reindexed.

FORCE

Recreate indexes forcedly. Without this keyword REINDEX does nothing unless target indexes are invalidated.

Outputs

`REINDEX`

Message returned if the table is successfully reindexed.

Description

REINDEX is used to recover corrupted system indexes. In order to run REINDEX command, postmaster must be shut down and stand-alone Postgres should be started instead with options -O and -P (an option to ignore system indexes). Note that we couldn't rely on system indexes for the recovery of system indexes.

Usage

Recreate the table `mytable`:

```
REINDEX TABLE mytable;
```

Some more examples:

```
REINDEX DATABASE my_database FORCE;
REINDEX INDEX my_index;
```

Compatibility

SQL92

There is no **REINDEX** in SQL92.

RESET

Name

`RESET` Restores run-time parameters to default values

Synopsis

```
RESET variable
```

Inputs

variable

The name of a run-time parameter. See *SET* for a list.

Description

RESET restores run-time parameters to their default values. Refer to *SET* for details. **RESET** is an alternate form for

```
SET variable TO DEFAULT
```

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 Postgres extension.

REVOKE

Name

REVOKE — Revokes access privilege from a user, a group or all users.

Synopsis

```
REVOKE privilege [, ...]
    ON object [, ...]
    FROM { PUBLIC | GROUP groupname | username }
```

Inputs

privilege

The possible privileges are:

SELECT

Privilege to access all of the columns of a specific table/view.

INSERT

Privilege to insert data into all columns of a specific table.

UPDATE

Privilege to update all columns of a specific table.

DELETE

Privilege to delete rows from a specific table.

RULE

Privilege to define rules on table/view. (See *CREATE RULE*).

ALL

Rescind all privileges.

object

The name of an object from which to revoke access. The possible objects are:
table
view
sequence

group

The name of a group from whom to revoke privileges.

username

The name of a user from whom revoke privileges. Use the PUBLIC keyword to specify all users.

PUBLIC

Rescind the specified privilege(s) for all users.

Outputs

`CHANGE`

Message returned if successfully.

`ERROR`

Message returned if object is not available or impossible to revoke privileges from a group or users.

Description

REVOKE allows creator of an object to revoke permissions granted before, from all users (via PUBLIC) or a certain user or group.

Notes

Refer to psql \z command for further information about permissions on existing objects:

```
Database    = lusitania
+------------------+---------------------------------------------+
|  Relation        |        Grant/Revoke Permissions             |
+------------------+---------------------------------------------+
| mytable          | {"=rw","miriam=arwR","group todos=rw"}      |
+------------------+---------------------------------------------+
Legend:
     uname=arwR -- privileges granted to a user
     group gname=arwR -- privileges granted to a GROUP
     =arwR -- privileges granted to PUBLIC

     r -- SELECT
     w -- UPDATE/DELETE
     a -- INSERT
     R -- RULE
     arwR -- ALL
```

Tip: Currently, to create a GROUP you have to insert data manually into table pg_group as:

```
INSERT INTO pg_group VALUES ('todos');
CREATE USER miriam IN GROUP todos;
```

Usage

Revoke insert privilege from all users on table `films`:

```
REVOKE INSERT ON films FROM PUBLIC;
```

Revoke all privileges from user `manuel` on view `kinds`:

```
REVOKE ALL ON kinds FROM manuel;
```

Compatibility

SQL92

The SQL92 syntax for **REVOKE** has additional capabilities for rescinding privileges, including those on individual columns in tables:

```
REVOKE { SELECT | DELETE | USAGE | ALL PRIVILEGES } [, ...]
    ON object
    FROM { PUBLIC | username [, ...] } { RESTRICT | CASCADE }
REVOKE { INSERT | UPDATE | REFERENCES } [, ...] [ ( column [, ...] ) ]
    ON object
    FROM { PUBLIC | username [, ...] } { RESTRICT | CASCADE }
```

Refer to *GRANT* for details on individual fields.

```
REVOKE GRANT OPTION FOR privilege [, ...]
    ON object
    FROM { PUBLIC | username [, ...] } { RESTRICT | CASCADE }
```

Rescinds authority for a user to grant the specified privilege to others. Refer to *GRANT* for details on individual fields.

The possible objects are:

[TABLE] table/view
CHARACTER SET character-set
COLLATION collation
TRANSLATION translation
DOMAIN domain

If user1 gives a privilege WITH GRANT OPTION to user2, and user2 gives it to user3 then user1 can revoke this privilege in cascade using the CASCADE keyword.

If user1 gives a privilege WITH GRANT OPTION to user2, and user2 gives it to user3, then if user1 tries to revoke this privilege it fails if he specify the RESTRICT keyword.

ROLLBACK

Name

`ROLLBACK` Aborts the current transaction

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ]
```

Inputs

None.

Outputs

`ABORT`

Message returned if successful.

`NOTICE: 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` Retrieves 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 ] [, ...] ]
    [ FOR UPDATE [ OF tablename [, ...] ] ]
    [ LIMIT { count | ALL } [ { OFFSET | , } start ]]

where from_item can be:

[ ONLY ] table_name [ * ]
    [ [ AS ] alias [ ( column_alias_list ) ] ]
|
( select )
    [ AS ] alias [ ( column_alias_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, 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, FOR UPDATE, and LIMIT clauses (even those can be used when the select is parenthesized).

FROM items can contain:

table_name

The name 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 as of Postgres 7.1 this is the default behavior. (In releases before 7.1, ONLY was the default behavior.)

alias

A substitute name for the preceding *table_name*. An alias is used for brevity or to eliminate ambiguity for self-joins (where the same table is scanned multiple times). 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.

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 FOR UPDATE clause allows the SELECT statement to perform exclusive locking of selected rows.

The LIMIT clause allows a subset of the rows produced by the query to be returned to the user. (See *LIMIT Clause*.)

You must have SELECT privilege to a table to read its values (See the **GRANT/REVOKE** statements).

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 Postgres 7.1, this was the default result, and adding sub-tables was done by appending ***** to the table name. This old behaviour 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.

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 NULLs 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 proper 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 keyword 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 >.

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, FOR UPDATE, or LIMIT 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 SELECTs 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, FOR UPDATE, or LIMIT 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, FOR UPDATE, or LIMIT 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 ]
    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 specified ORDER BY.

As of Postgres 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 give 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.

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
```

Compatibility

Extensions

Postgres 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 constant expressions:

```
SELECT 2+2;

 ?column?
----------
        4
```

Some other DBMSes 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 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, Postgres 7.1 and later will warn if the implicit-FROM feature is used in a query that also contains an explicit FROM clause.

SQL92

SELECT Clause

In the SQL92 standard, the optional keyword "AS" is just noise and can be omitted without affecting the meaning. The Postgres parser requires this keyword when renaming 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. Postgres extends each of these clauses to allow the other choice as well (but it uses the standard's interpretation if there is ambiguity). Postgres 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 Postgres.

SELECT INTO

Name

`SELECT INTO` Creates a new table from the results of a SELECT

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 ] [, ...] ]
    [ FOR UPDATE [ OF tablename [, ...] ] ]
    [ LIMIT { count | ALL } [ { OFFSET | , } start ]]

where from_item can be:

[ ONLY ] table_name [ * ]
    [ [ AS ] alias [ ( column_alias_list ) ] ]
|
( select )
    [ AS ] alias [ ( column_alias_list ) ]
|
from_item [ NATURAL ] join_type from_item
    [ ON join_condition | USING ( join_column_list ) ]
```

Inputs

TEMPORARY
TEMP

If TEMPORARY or TEMP is specified, the output table is created only within this session, and is automatically dropped on session exit. Existing permanent tables with the same name are not visible (in this session) while the temporary table exists. Any indexes created on a temporary table are automatically temporary as well.

`new_table`

The name of the new table to be created. This table must not already exist. However, a temporary table can be created that has the same name as an existing permanent table.

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 datatypes 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 Postgres 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` Set run-time parameters

Synopsis

```
SET variable { TO | = } { value | 'value' | DEFAULT }
SET TIME ZONE { 'timezone' | LOCAL | DEFAULT }
```

Inputs

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. The following parameters can be altered:

CLIENT_ENCODING

NAMES

Sets the multibyte client encoding. The specified encoding must be supported by the backend.

This option is only available if Postgres is build with multibyte support.

DATESTYLE

Choose the date/time representation style. Two separate settings are made: 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.

Postgres

Use traditional Postgres format.

German

Use `dd.mm.yyyy` for numeric date representations.

The following two options determine both a substyle of the SQL and Postgres 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.

Date format initialization may be done by:

Setting the 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.
Running postmaster using the option `-o -e` to set dates to the `European` convention.

The `DateStyle` option is really only intended for porting applications. To format your date/time values to choice, use the `to_char` family of functions.

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. This product will silently overflow if a number outside the range is used.

The seed can also be set by invoking the `setseed` SQL function:

```
SELECT setseed(value);
```

SERVER_ENCODING

Sets the multibyte server encoding.

This option is only available if Postgres was built with multibyte support.

TIME ZONE
TIMEZONE

The possible values for time zone depends on your operating system. For example, on Linux `/usr/share/zoneinfo` contains the database of time zones.

Here are some valid values for time zone:

PST8PDT

Set the time zone for California.

Portugal

Set time zone for Portugal.

’Europe/Rome’

Set time zone for Italy.

LOCAL
DEFAULT

Set the time zone to your local time zone (the one that your operating system defaults to).

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.

An extended list of other run-time parameters can be found in the *Administrator’s Guide*.

Use *SHOW* to show the current setting of a parameters.

Diagnostics

`SET VARIABLE`

Message returned if successful.

`ERROR: not a valid option name: name`

The parameter you tried to set does not exist.

`ERROR: permission denied`

You must be a superuser to have access to certain settings.

`ERROR: name can only be set at start-up`

Some parameters are fixed once the server is started.

Examples

Set the style of date to traditional Postgres with European conventions:

```
SET DATESTYLE TO Postgres,European;
```

Set the time zone for Berkeley, California, using double quotes to preserve the uppercase attributes of the time zone specifier (note that the date/time format is ISO here):

```
SET TIME ZONE "PST8PDT";
SELECT CURRENT_TIMESTAMP AS today;

         today
------------------------
 1998-03-31 07:41:21-08
```

Set the time zone for Italy (note the required single or double quotes to handle the special characters):

```
SET TIME ZONE 'Europe/Rome';
SELECT CURRENT_TIMESTAMP AS today;

         today
------------------------
 1998-03-31 17:41:31+02
```

Compatibility

SQL92

The second syntax shown above (`SET TIME ZONE`) attempts to mimic SQL92. However, SQL allows only numeric time zone offsets. All other parameter settings as well as the first syntax shown above are a Postgres extension.

SET CONSTRAINTS

Name

`SET CONSTRAINTS` Set the constraint mode of the current SQL-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.

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 CONSTRAINT is defined in SQL92 and SQL99.

SET TRANSACTION

Name

`SET TRANSACTION` Set the characteristics of the current SQL-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 DML 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 DML 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.

Compatibility

SQL92, SQL99

SERIALIZABLE is the default level in SQL. Postgres does not provide the isolation levels `READ UNCOMMITTED` and `REPEATABLE READ`. Because of multi-version 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 Postgres.

SHOW

Name

`SHOW` Shows run-time parameters

Synopsis

```
SHOW name
```

Inputs

name

The name of a run-time parameter. See *SET* for a list.

Description

SHOW will display the current setting of a run-time parameter. These variables can be set using the **SET** statement or are determined at server start.

Diagnostics

`ERROR: not a valid option name: name`

Message returned if *variable* does not stand for an existing parameter.

`ERROR: permission denied`

You must be a superuser to be allowed to see certain settings.

`NOTICE: Time zone is unknown`

If the TZ or PGTZ environment variable is not set.

Examples

Show the current `DateStyle` setting:

```
SHOW DateStyle;
NOTICE:  DateStyle is ISO with US (NonEuropean) conventions
```

Show the current genetic optimizer (`geqo`) setting:

```
SHOW GEQO;
NOTICE:  geqo = true
```

Compatibility

The **SHOW** command is a Postgres extension.

TRUNCATE

Name

`TRUNCATE` Empty a table

Synopsis

```
TRUNCATE [ TABLE ] name
```

Inputs

`name`

The name of the table to be truncated.

Outputs

`TRUNCATE`

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 effective on large tables.

Usage

Truncate the table `bigtable`:

```
TRUNCATE TABLE bigtable;
```

Compatibility

SQL92

There is no **TRUNCATE** in SQL92.

UNLISTEN

Name

`UNLISTEN` Stop listening for 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 Postgres session as a listener on the notify condition *`notifyname`*. The special condition wildcard "*" cancels all listener registrations for the current session.

NOTIFY contains a more extensive discussion of the use of **LISTEN** and **NOTIFY**.

Notes

`notifyname` need not be a valid class name but can be any string valid as a name up to 32 characters long.

The backend does not complain if you UNLISTEN something you were not listening for. Each backend will automatically execute **UNLISTEN *** when exiting.

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` Replaces values of columns in a table

Synopsis

```
UPDATE [ ONLY ] table SET col = expression [, ...]
    [ FROM fromlist ]
    [ WHERE condition ]
```

Inputs

table

The name of an existing table.

column

The name of a column in *table*.

expression

A valid expression or value to assign to column.

fromlist

A Postgres non-standard extension to allow columns from other tables to appear in the WHERE condition.

condition

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

Outputs

`UPDATE #`

Message returned if successful. The # means the number of rows updated. If # is 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 Clean and analyze a Postgres database

Synopsis

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

Inputs

VERBOSE

Prints a detailed vacuum activity report for each table.

ANALYZE

Updates column statistics used by the optimizer to determine the most efficient way to execute a query.

table

The name of a specific table to vacuum. Defaults to all tables.

column

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

Outputs

VACUUM

The command has been accepted and the database is being cleaned.

NOTICE: --Relation *table*--

The report header for *table*.

```
NOTICE: Pages 98: Changed 25, Reapped 74, Empty 0, New 0; Tup 1000: Vac
3000, Crash 0, UnUsed 0, MinLen 188, MaxLen 188; Re-using: Free/Avail.
Space 586952/586952; EndEmpty/Avail. Pages 0/74. Elapsed 0/0 sec.
```

The analysis for *table* itself.

```
NOTICE: Index index: Pages 28; Tuples 1000: Deleted 3000. Elapsed 0/0 sec.
```

The analysis for an index on the target table.

Description

VACUUM serves two purposes in Postgres as both a means to reclaim storage and also a means to collect information for the optimizer.

VACUUM opens every table in the database, cleans out records from rolled back transactions, and updates statistics in the system catalogs. The statistics maintained include the number of tuples and number of pages stored in all tables.

VACUUM ANALYZE collects statistics representing the dispersion of the data in each column. This information is valuable when several query execution paths are possible.

Running **VACUUM** periodically will increase the speed of the database in processing user queries.

Notes

The open database is the target for **VACUUM**.

We recommend that active production databases be **VACUUM**-ed nightly, in order to remove expired rows. After copying a large table into Postgres or after deleting a large number of records, it may be a good idea to issue a **VACUUM ANALYZE** query. This will update the system catalogs with the results of all recent changes, and allow the Postgres query optimizer to make better choices in planning user queries.

Usage

The following is partial example from running **VACUUM** on a table in the regression database:

```
regression=> vacuum verbose analyze onek;
NOTICE:  --Relation onek--
NOTICE:  Pages 98: Changed 25, Reapped 74, Empty 0, New 0;
         Tup 1000: Vac 3000, Crash 0, UnUsed 0, MinLen 188, MaxLen 188;
         Re-using: Free/Avail. Space 586952/586952; EndEmpty/Avail. Pages
0/74.
         Elapsed 0/0 sec.
NOTICE:  Index onek_stringu1: Pages 28; Tuples 1000: Deleted 3000. Elapsed
0/0 sec.
VACUUM
```

Compatibility

SQL92

There is no **VACUUM** statement in SQL92.

II. PostgreSQL Client Applications

This is reference information for Postgres client applications and utilities.

createdb

Name

`createdb` Create a new Postgres database

Synopsis

createdb [*options*...] [*dbname*] [*description*]

Inputs

-h, --host *host*

Specifies the hostname 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.

-p, --port *port*

Specifies the Internet TCP/IP port or the local Unix domain socket file extension on which the **postmaster** is listening for connections.

-U, --username *username*

Username to connect as.

-W, --password

Force password prompt.

-e, --echo

Echo the queries that createdb generates and sends to the backend.

-q, --quiet

Do not display a response.

-D, --location *datadir*

Specifies the alternative location for the database. See also initlocation.

-T, --template *template*

Specifies the template database from which to build this database.

-E, --encoding *encoding*

Specifies the character encoding scheme to be used in this database.

dbname

Specifies the name of the database to be created. The name must be unique among all Postgres 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.

The options `-h`, `-p`, `-U`, `-W`, and `-e` are passed on literally to psql. The options `-D`, `-T`, and `-E` are converted into options for the underlying SQL command *CREATE DATABASE*; see there for more information about them.

Outputs

`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.

Description

createdb creates a new Postgres database. The user who executes this command becomes the database owner.

createdb is a shell script wrapper around the SQL command *CREATE DATABASE* via the Postgres 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.

Usage

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 postmaster 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
```

createuser

Name

`createuser` Create a new Postgres user

Synopsis

createuser [*`options`*...] [*`username`*]

Inputs

-h, --host *`host`*

Specifies the hostname 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.

-p, --port *`port`*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections.

-e, --echo

Echo the queries that createdb generates and sends to the backend.

-q, --quiet

Do not display a response.

-d, --createdb

Allows the new user to create databases.

-D, --no-createdb

Forbids the new user to create databases.

-a, --adduser

Allows the new user to create other users.

-A, --no-adduser

Forbids the new user to create other users.

-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.

-i, --sysid *`uid`*

Allows you to pick a non-default user id for the new user. This is not necessary, but some people like it.

username

Specifies the name of the Postgres user to be created. This name must be unique among all Postgres users.

You will be prompted for a name and other missing information if it is not specified on the command line.

The options `-h`, `-p`, and `-e`, are passed on literally to psql. The psql options `-U` and `-W` are available as well, but their use can be confusing in this context.

Outputs

`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.

Description

createuser creates a new Postgres user. Only users with `usesuper` set in the `pg_shadow` table can create new Postgres users.

createuser is a shell script wrapper around the SQL command *CREATE USER* via the Postgres interactive terminal psql. Thus, there is nothing special about creating 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.

Usage

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 postmaster 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
```

dropdb

Name

`dropdb` Remove an existing Postgres database

Synopsis

dropdb [*options*...] *dbname*

Inputs

-h, --host *host*

Specifies the hostname 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.

-p, --port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections.

-U, --username *username*

Username to connect as.

-W, --password

Force password prompt.

-e, --echo

Echo the queries that dropdb generates and sends to the backend.

-q, --quiet

Do not display a response.

-i, --interactive

Issues a verification prompt before doing anything destructive.

dbname

Specifies the name of the database to be removed. The database must be one of the existing Postgres databases in this installation.

The options `-h`, `-p`, `-U`, `-W`, and `-e` are passed on literally to psql.

Outputs

`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.

Description

dropdb destroys an existing Postgres 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 Postgres 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.

Usage

To destroy the database `demo` on the default database server:

```
$ dropdb demo
DROP DATABASE
```

To destroy the database `demo` using the postmaster 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
```

dropuser

Name

`dropuser` Drops (removes) a Postgres user

Synopsis

dropuser [*options*...] [*username*]

Inputs

-h, --host *host*

Specifies the hostname 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.

-p, --port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections.

-e, --echo

Echo the queries that createdb generates and sends to the backend.

-q, --quiet

Do not display a response.

-i, --interactive

Prompt for confirmation before actually removing the user.

username

Specifies the name of the Postgres user to be removed. This name must exist in the Postgres installation. You will be prompted for a name if none is specified on the command line.

The options `-h`, `-p`, and `-e`, are passed on literally to psql. The psql options `-U` and `-W` are available as well, but they can be confusing in this context.

Outputs

`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.

Description

dropuser removes an existing Postgres user *and* the databases which that user owned. Only users with `usesuper` set in the `pg_shadow` table can destroy Postgres users.

dropuser is a shell script wrapper around the SQL command *DROP USER* via the Postgres 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.

Usage

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
```

ecpg

Name

`ecpg` Embedded SQL C preprocessor

Synopsis

```
ecpg [ -v ] [ -t ] [ -I include-path ] [ -o outfile ]  file1 [ file2 ] [
... ]
```

Inputs

ecpg accepts the following command line arguments:

-v

Print version information.

-t

Turn off auto-transaction mode.

-I *path*

Specify an additional include path. Defaults are `.`, `/usr/local/include`, the Postgres include path which is defined at compile time (default: `/usr/local/pgsql/lib`), and `/usr/include`.

-o

Specifies that ecpg should write all its output to outfile. If no such option is given the output is written to *name*`.c`, assuming the input file was named *name*`.pgc`. If the input file does have the expected `.pgc` suffix, then the output file will have `.pgc` appended to the input file name.

file

The files to be processed.

Outputs

ecpg will create a file or write to `stdout`.

return value

ecpg returns 0 to the shell on successful completion, -1 for errors.

Description

ecpg is an embedded SQL preprocessor for the C language and the Postgres. It enables development of C programs with embedded SQL code.

Linus Tolke (<linus@epact.se>) was the original author of ecpg (up to version 0.2). Michael Meskes (<meskes@debian.org>) is the current author and maintainer of ecpg. Thomas Good (<tomg@q8.nrnet.org>) is the author of the last revision of the ecpg man page, on which this document is based.

Usage

Preprocessing for Compilation

An embedded SQL source file must be preprocessed before compilation:

```
ecpg [ -d ] [ -o file ] file.pgc
```

where the optional `-d` flag turns on debugging. The `.pgc` extension is an arbitrary means of denoting ecpg source.

You may want to redirect the preprocessor output to a log file.

Compiling and Linking

Assuming the Postgres binaries are in `/usr/local/pgsql`, you will need to compile and link your preprocessed source file:

```
gcc -g -I /usr/local/pgsql/include [ -o file ] file.c -L
/usr/local/pgsql/lib -lecpg -lpq
```

Grammar

Libraries

The preprocessor will prepend two directives to the source:

```
#include <ecpgtype.h>
#include <ecpglib.h>
```

Variable Declaration

Variables declared within ecpg source code must be prepended with:

```
EXEC SQL BEGIN DECLARE SECTION;
```

Similarly, variable declaration sections must terminate with:

```
EXEC SQL END DECLARE SECTION;
```

> **Note:** Prior to version 2.1.0, each variable had to be declared on a separate line. As of version 2.1.0 multiple variables may be declared on a single line:
>
> ```
> char  foo(16), bar(16);
> ```

Error Handling

The SQL communication area is defined with:

```
EXEC SQL INCLUDE sqlca;
```

> **Note:** The `sqlca` is in lowercase. While SQL convention may be followed, i.e., using uppercase to separate embedded SQL from C statements, sqlca (which includes the sqlca.h header file) MUST be lowercase. This is because the EXEC SQL prefix indicates that this INCLUDE will be parsed by ecpg. ecpg observes case sensitivity (SQLCA.h will not be found). **EXEC SQL INCLUDE** can be used to include other header files as long as case sensitivity is observed.

The sqlprint command is used with the EXEC SQL WHENEVER statement to turn on error handling throughout the program:

```
EXEC SQL WHENEVER sqlerror sqlprint;
```

and

```
EXEC SQL WHENEVER not found sqlprint;
```

> **Note:** This is *not* an exhaustive example of usage for the **EXEC SQL WHENEVER** statement. Further examples of usage may be found in SQL manuals (e.g., 'The LAN TIMES Guide to SQL' by Groff and Weinberg).

Connecting to the Database Server

One connects to a database using the following:

```
EXEC SQL CONNECT TO dbname;
```

where the database name is not quoted. Prior to version 2.1.0, the database name was required to be inside single quotes.

Specifying a server and port name in the connect statement is also possible. The syntax is:

```
dbname[@server][:port]
```

or

```
<tcp|unix>:postgresql://server[:port][/dbname][?options]
```

Queries

In general, SQL queries acceptable to other applications such as psql can be embedded into your C code. Here are some examples of how to do that.

Create Table:

```
EXEC SQL CREATE TABLE foo (number int4, ascii char(16));
EXEC SQL CREATE UNIQUE index num1 on foo(number);
EXEC SQL COMMIT;
```

Insert:

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

Delete:

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

Singleton Select:

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

Select using Cursors:

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
```

```
EXEC SQL COMMIT;
```

Updates:

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```

Notes

There is no **EXEC SQL PREPARE** statement.

The complete structure definition MUST be listed inside the declare section.

See the `TODO` file in the source for some more missing features.

pgaccess

Name

`pgaccess` PostgreSQL graphical client

Synopsis

pgaccess [*dbname*]

Options

dbname

The name of an existing database to access.

Description

pgaccess provides a graphical interface for Postgres wherein you can manage your tables, edit them, define queries, sequences and functions.

pgaccess can:

- Open any database on a specified host at the specified port, username, and password.
- Execute *VACUUM*.
- Save preferences in the `~/.pgaccessrc` file.

For tables, pgaccess can:

- Open multiple tables for viewing, with a configurable number of rows shown.
- Resize columns by dragging the vertical grid lines.
- Wrap text in cells.
- Dynamically adjust row height when editing.
- Save table layout for every table.

Import/export to external files (SDF, CSV).
Use filter capabilities; enter filters like `price > 3.14`.
Specify sort order; enter manually the sort field(s).
Edit in place; double click the text you want to change.
Delete records; point to the record, press **Delete** key.
Add new records; save new row with right-button click.
Create tables with an assistant.
Rename and delete (drop) tables.
Retrieve information on tables, including owner, field information, indices.

For queries, pgaccess can:
Define, edit and store user-defined queries.
Save view layouts.
Store queries as views.
Execute with optional user input parameters, e.g.,

```
select * from invoices where year=[parameter "Year of selection"]
```

View any select query result.
Run action queries (insert, update, delete).
Construct queries using a visual query builder with drag & drop support, table aliasing.

For sequences, pgaccess can:
Define new instances.
Inspect existing instances.
Delete.

For views, pgaccess can:
Define them by saving queries as views.
View them, with filtering and sorting capabilities.
Design new views.
Delete (drop) existing views.

For functions, pgaccess can:
Define.
Inspect.
Delete.

For reports, pgaccess can:
Generate simple reports from a table (beta stage).
Change font, size, and style of fields and labels.
Load and save reports from the database.
Preview tables, sample Postscript print.

For forms, pgaccess can:
Open user-defined forms.
Use a form design module.

Access record sets using a query widget.

For scripts, pgaccess can:

Define.

Modify.

Call user defined scripts.

Notes

pgaccess is written in Tcl/Tk. Your PostgreSQL installation needs to be built with Tcl support for pgaccess to be available.

pgadmin

Name

`pgadmin` Postgres database management and design tool for Windows 95/98/NT

Synopsis

`pgadmin [ `*`datasourcename`*` [ `*`username`*` [ `*`password`*` ] ] ]`

Inputs

`datasourcename`

The name of an existing Postgres ODBC System or User Data Source.

`username`

A valid username for the specified *`datasourcename`*.

`password`

A valid password for the specified *`datasourcename`* and *`username`*.

Outputs

Description

pgadmin is a general purpose tool for designing, maintaining, and administering Postgres databases. It runs under Windows 95/98 and NT.

Features include:

Arbitrary SQL entry.

Info Browsers and 'Creators' for databases, tables, indexes, sequences, views, triggers, functions and languages.

User, Group and Privilege configuration dialogues.

Revision tracking with upgrade script generation.

Configuration of Microsoft MSysConf table.

Data Import and Export Wizards.

Database Migration Wizard.

Predefined reports on databases, tables, indexes, sequences, languages and views.

pgadmin is distributed separately from Postgres and may be downloaded from http://www.pgadmin.freeserve.co.uk

pg_config

Name

`pg_config` Provides information about the installed version of PostgreSQL

Synopsis

pg_config {--bindir | --includedir | --libdir | --configure | --version...}

Description

The pg_config utility provides 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 in order to find the respective header files and libraries.

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 and C++ header files.

--libdir

Print the location of object code libraries.

--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.

pg_dump

Name

`pg_dump` Extract a Postgres database into a script file or other archive file

Synopsis

pg_dump [-a | -s] [-b] [-c] [-C] [-d | -D] [-f *file*] [-F *format*] [-i] [-n | -N] [-o] [-O] [-R] [-S] [-t *table*] [-v] [-x] [-Z *0...9*] [-h *host*] [-p *port*] [-u] *dbname*

Description

pg_dump is a utility for dumping out a Postgres database into a script or archive file containing query commands. The script files are in text format and can be used to reconstruct the database, even on other machines and other architectures. The archive files, new with version 7.1, contain enough 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 also designed to be portable across architectures.

pg_dump will produce the queries necessary to re-generate all user-defined types, functions, tables, indices, aggregates, and operators. In addition, all the data is copied out in text format so that it can be readily copied in again, as well as imported into tools for editing.

pg_dump is useful for dumping out the contents of a database to move from one Postgres installation to another. After running **pg_dump**, one should examine the output for any warnings, especially in light of the limitations listed below.

When used with one of the alternate file formats and combined with **pg_restore**, it 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. See the pg_restore documentation for details.

Options

pg_dump accepts the following command line arguments. (Long option forms are only available on some platforms.)

dbname

Specifies the name of the database to be extracted.

-a
--data-only

Dump only the data, not the schema (definitions).

-b
--blobs

Dump data and BLOB data.

-c
--clean

Dump commands to clean (drop) the schema prior to (the commands for) creating it.

-C
--create

For plain text (script) output, include commands to create the database itself.

-d
--inserts

Dump data as proper **INSERT** commands (not **COPY**). This will make restoration very slow.

-D
--attribute-inserts

Dump data as **INSERT** commands with explicit column names. This will make restoration very slow.

-f *file*
--file=*file*

Send output to the specified file.

-F *format*
--format=*format*

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. Since **pg_dump** knows a great deal about system catalogs, any given version of **pg_dump** is only intended to work with the corresponding release of the database server. Use this option if you need to override the version check (and if **pg_dump** then fails, don't say you weren't warned).

-n
--no-quotes

Suppress double quotes around identifiers unless absolutely necessary. This may cause trouble loading this dumped data if there are reserved words used for identifiers. This was the default behavior for **pg_dump** prior to version 6.4.

-N
--quotes

Include double quotes around identifiers. This is the default.

-o
--oids

Dump object identifiers (OIDs) for every table.

-O
--no-owner

In plain text output mode, do not set object ownership to match the original database. Typically, **pg_dump** issues (**psql**-specific) **\connect** statements to set ownership of schema elements.

-R
--no-reconnect

In plain text output mode, prohibit **pg_dump** from issuing any **\connect** statements.

-s
--schema-only

Dump only the schema (definitions), no data.

-S *username*
--superuser=*username*

Specify the superuser user name to use when disabling triggers and/or setting ownership of schema elements.

-t *table*
--table=*table*

Dump data for *table* only.

-v
--verbose

Specifies verbose mode.

-x
--no-acl

Prevent dumping of ACLs (grant/revoke commands) and table ownership information.

-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).

pg_dump also accepts the following command line arguments for connection parameters:

-h *host*
--host=*host*

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.

-p *port*
--port=*port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the **postmaster** is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

-u

Use password authentication. Prompts for *username* and *password*.

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 **postmaster** process on the specified host and port. If you see this message, ensure that the **postmaster** is running on the proper host and that you have specified the proper port.

`dumpSequence(table): SELECT failed`

You do not have permission to read the database. Contact your Postgres site administrator.

Note: pg_dump internally executes **SELECT** statements. If you have problems running **pg_dump**, make sure you are able to select information from the database using, for example, psql.

Notes

pg_dump has a few limitations. The limitations mostly stem from difficulty in extracting certain meta-information from the system catalogs.

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.

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 BLOBs to a `tar` file:

```
$ pg_dump -Ft -b mydb > db.tar
```

To reload this database (with BLOBs) to an existing database called newdb:

```
$ pg_restore -d newdb db.tar
```

See Also

pg_dumpall, pg_restore , psql, *PostgreSQL Administrator's Guide*

pg_dumpall

Name

`pg_dumpall` Extract all databases into a script file

Synopsis

pg_dumpall [-c | --clean] [-h *host*] [-p *port*] [-g | --globals-only]

Description

pg_dumpall is a utility for writing out (dumping) all Postgres 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.

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 ouput. Shell operators should be used to redirect it into a file.

Options

pg_dumpall accepts the following command line arguments:

-c, --clean

Clean (drop) database before creating schema.

-h *host*

Specifies the hostname 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.

-g, --globals-only

Only dump global objects (users and groups), no databases.

Any other command line parameters are passed to the underlying pg_dump calls. This is useful to control some aspects of the output format, but some options such as `-f`, `-t`, and *dbname* should be avoided.

Usage

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 Postgres database from an archive file created by **pg_dump**

Synopsis

pg_restore [-a] [-c] [-C] [-d *dbname*] [-f *archive-file*] [-F *format*] [-i *index*] [-l] [-L *contents-file*] [-N | -o | -r] [-O] [-P *function-name*] [-R] [-s] [-S] [-t *table*] [-T *trigger*] [-v] [-x] [-h *host*] [-p *port*] [-u] [*archive-file*]

Description

pg_restore is a utility for restoring a Postgres database dumped by pg_dump in one of the non-plain-text formats.

The archive files, new with the 7.1 release, contain enough 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_dump** will produce the queries necessary to re-generate all user-defined types, functions, tables, indices, aggregates, and operators. In addition, all the data is copied out (in text format for scripts) so that it can be readily copied in again.

pg_restore reads the archive file and outputs the appropriate SQL in the required order based on the command parameters. Obviously, it can not restore information that is not present in the dump file; so if the dump is made using the dump data as **INSERT**s option, **pg_restore** will not be able to load the data using **COPY** statements.

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.

To reorder the items, it is first necessary to dump the contents of the archive:

```
$ pg_restore archive.file -l > archive.list
```

This 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 archive.file -L archive.list
```

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, and no `-f` option is specified, then the standard input is used.

-a
--data-only

Restore only the data, no schema (definitions).

-c
--clean

Clean (drop) schema prior to create.

-C
--create

Include SQL to create the schema.

-d *dbname*
--dbname=*dbname*

Connect to database *dbname* and restore directly into the database. BLOBs can only be restored by using a direct database connection.

-f *filename*
--file=*filename*

Specify output file for generated script. (Use with the `-l` option.) 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 *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`
--function=`function-name`

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 indices) will be restored at the end of the process irrespective of their OIDs. This option is the default.

-R
--no-reconnect

Prohibit **pg_restore** from issuing any

```
\connect
```

statements or reconnecting to the database if directly connected.

-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 and/or setting ownership of schema elements. By default, **pg_restore** will use the current user name if it is a superuser.

-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-acl

Prevent restoration of ACLs (grant/revoke commands).

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 **postmaster** 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 **postmaster** is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

-u

Use password authentication. Prompts for user name and password.

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 **postmaster** process on the specified host and port. If you see this message, ensure that the **postmaster** is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

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

The limitations of **pg_restore** are detailed below.

When restoring data to a 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 BLOBs for a single table. If an archive contains BLOBs, then all BLOBs will be restored.

See the pg_dump documentation for details on limitation 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 BLOBs to a `tar` file:

```
$ pg_dump -Ft -b mydb > db.tar
```

To reload this database (with BLOBs) to an existing database called newdb:

```
$ pg_restore -d newdb db.tar
```

See Also

pg_dump , pg_dumpall, psql, *PostgreSQL Administrator's Guide*

psql

Name

`psql` Postgres interactive terminal

Synopsis

`psql [ options ] [ dbname [ user ] ]`

Summary

psql is a terminal-based front-end to Postgres. It enables you to type in queries interactively, issue them to Postgres, 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.

Description

Connecting To A Database

psql is a regular Postgres client application. In order to connect to a database you need to know the name of your target database, the hostname 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 username, as is the default database name. Note that you can't just connect to any database under any username. 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, 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*.

psql 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 must 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`, `\0digits`, and `\0xdigits` (the character with the given decimal, octal, or hexadecimal code).

If an unquoted argument begins with a colon (`:`), it is taken as a variable and the value of the variable is taken as the argument instead.

Arguments that are quoted in backticks (`‘`) are taken as a command line that is passed to the shell. The output of the command (with a trailing newline removed) is taken as the argument value. The above escape sequences also apply in backticks.

Some commands take the name of 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. For all other commands double quotes are not special and will become part of 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.

`\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 username, 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`* [`with oids`] { `from` | `to` } *`filename`* | stdin | stdout [`using delimiters` '*`characters`*'] [`with 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, and consequently requiring backend access and special user privilege, as well as being bound to the file system accessible by the backend, psql reads or writes the file and routes the data between the backend and the local file system.

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 Postgres.

`\d` *`relation`*

Shows all columns of *`relation`* (which could be a table, view, index, or sequence), their types, and any special attributes such as `NOT NULL` or defaults, if any. If the relation is, in fact, a table, any defined indices are also listed. If the relation is a view, the view definition is also shown.

The command form `\d+` is identical, but any comments associated with the table columns are shown as well.

> **Note:** If **\d** is called without any arguments, 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` [*`object`*]

Shows the descriptions of *`object`* (which can be a regular expression), or of all objects if no argument is given. (Object covers aggregates, functions, operators, types, relations (tables, views, indices, sequences, large objects), rules, and triggers.) For example:

```
=> \dd version
              Object descriptions
  Name   |   What   |        Description
---------+----------+---------------------------
 version | function | PostgreSQL version string
(1 row)
```

Descriptions for objects can be generated with the **COMMENT ON** SQL command.

> **Note:** Postgres stores the object descriptions in the pg_description system table.

`\df [` *`pattern`* `]`

Lists available functions, together with their argument and return types. If *`pattern`* (a regular expression) is specified, only matching functions are shown. If the form `\df+` is used, additional information about each function, including language and description, is shown.

`\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 them in any order to obtain a listing of them, together with who the owner is.

If *`pattern`* is specified, it is a regular expression that restricts the listing to those objects whose name matches. If one appends a + to the command name, each object is listed with its associated description, if any.

`\dl`

This is an alias for **\lo_list**, which shows a list of large objects.

`\do [ name ]`

Lists available operators with their operand and return types. If *name* is specified, only operators with that name will be shown.

`\dp [ pattern ]`

This is an alias for **\z** which was included for its greater mnemonic value (display permissions).

`\dT [ pattern ]`

Lists all data types or only those that match *pattern*. The command form `\dT+` shows extra information.

`\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 Postgres 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 Postgres 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 Postgres *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).

`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 list of a pager to do table output. If the environment variable PAGER is set, the output is piped to the specified program. Otherwise `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 and when not to.

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:** As of psql version 7.0 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`*.

`\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 tables in the database with their appropriate access permissions listed. If an argument is given it is taken as a regular expression which limits the listing to those tables which match it.

```
test=> \z
Access permissions for database "test"
 Relation |           Access permissions
----------+-------------------------------------
 my_table | {"=r","joe=arwR", "group staff=ar"}
(1 row )
```

Read this as follows:

`"=r"`: `PUBLIC` has read (**SELECT**) permission on the table.

`"joe=arwR"`: User `joe` has read, write (**UPDATE**, **DELETE**), append (**INSERT**) permissions, and permission to create rules on the table.

`"group staff=ar"`: Group `staff` has **SELECT** and **INSERT** permission.

The commands *GRANT* and *REVOKE* are used to set access 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.

Command-line Options

If so configured, psql understands both standard Unix short options, and GNU-style long options. The latter are not available on all systems.

-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, --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 parseable 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*

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, --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**.

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, --field-separator *separator*

Use *separator* as the field separator. This is equivalent to **\pset fieldsep** or **\f**.

-h, --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, --output *filename*

Put all query output into file *filename*. This is equivalent to the command **\o**.

-p, --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, --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

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, --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-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*

Connects to the database as the user *username* instead of the default. (You must have permission to do so, of course.)

-v, --variable, --set *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**.

As of version 7.0, 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.

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 runtime 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 Postgres 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 Postgres 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 Postgres 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 hostname (with domain name) of the database server (or localhost if hostname information is not available).

`%m`

The hostname of the database server, truncated after the first dot.

`%>`

The port number at which the database server is listening.

`%n`

The username 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.

Miscellaneous

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.

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).

GNU readline

psql supports the readline and history libraries 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.)

If you have the readline library installed but psql does not seem to use it, you must make sure that Postgres's top-level `configure` script finds it. `configure` needs to find both the library `libreadline.a` (or a shared library equivalent) *and* the header files `readline.h` and `history.h` (or `readline/readline.h` and `readline/history.h`) in appropriate directories. If you have the library and header files installed in an obscure place you must tell `configure` about them, for example:

```
$ ./configure --with-includes=/opt/gnu/include --with-libs=/opt/gnu/lib
...
```

Then you have to recompile psql (not necessarily the entire code tree).

The GNU readline library can be obtained from the GNU project's FTP server at ftp://ftp.gnu.org.

Examples

> **Note:** This section only shows a few examples specific to psql. If you want to learn SQL or get familiar with Postgres, 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
```

Appendix

Bugs and Issues

In some earlier life psql allowed the first argument to start directly after the (single-letter) command. For compatibility this is still supported to some extent but I am not going to explain the details here as this use is discouraged. But 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.

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 PQexec: you gotta get out of a COPY state yourself, simply reset the connection by entering `\c - -`.

pgtclsh

Name

`pgtclsh` PostgreSQL Tcl shell client

Synopsis

pgtclsh [*`filename`* [*`arguments`*...]]

Description

pgtclsh is a Tcl shell interface extended with Postgres 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 Postgres 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 Postgres 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 Postgres functions is available as pgtclsh .

See Also

pgtclsh , *PostgreSQL Programmer's Guide* (description of `libpgtcl`) , tclsh , wish

vacuumdb

Name

`vacuumdb` Clean and analyze a Postgres database

Synopsis

vacuumdb [*connection-options*...] [[-d] *dbname*] [--analyze | -z] [--verbose | -v] [--table '*table* [(*column* [,...])]']
vacuumdb [*connection-options*...] [--all | -a] [--analyze | -z] [--verbose | -v]

Inputs

vacuumdb accepts the following command line arguments:

-d *dbname*
--dbname *dbname*

Specifies the name of the database to be cleaned or analyzed.

-z
--analyze

Calculate statistics on the database for use by the optimizer.

-a
--alldb

Vacuum all databases.

-v
--verbose

Print detailed information during processing.

-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.

vacuumdb also accepts the following command line arguments for connection parameters:

-h *host*
--host *host*

Specifies the hostname 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.

-p *port*
--port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections.

-U *username*
--username *username*

Username to connect as.

-W
--password

Force password prompt.

-e
--echo

Echo the commands that vacuumdb generates and sends to the backend.

-q
--quiet

Do not display a response.

Outputs

`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.

Description

vacuumdb is a utility for cleaning a Postgres database. vacuumdb will also generate internal statistics used by the Postgres query optimizer.

vacuumdb is a shell script wrapper around the backend command *VACUUM* via the Postgres 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.

Usage

To clean the database `test`:

```
$ vacuumdb test
```

To analyze for the optimzer a database named `bigdb`:

```
$ vacuumdb --analyze bigdb
```

To analyze a single column `bar` in table `foo` in a database named `xyzzy` for the optimizer:

```
$ vacuumdb --analyze --verbose --table ’foo(bar)’ xyzzy
```

III. PostgreSQL Server Applications

This is reference information for Postgres server applications and support utilities.

createlang

Name

`createlang` Add a new programming language to a Postgres database

Synopsis

createlang [*`connection-options`*...] [*`langname`*] *`dbname`*
createlang [*`connection-options`*...] --list | -l *`dbname`*

Inputs

createlang accepts the following command line arguments:

`langname`

Specifies the name of the backend programming language to be defined. createlang will prompt for *`langname`* if it is not specified on the command line.

-d, --dbname *`dbname`*

Specifies which database the language should be added.

-l, --list

Shows a list of already installed languages in the target database (which must be specified).

createlang also accepts the following command line arguments for connection parameters:

-h, --host *`host`*

Specifies the hostname 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.

-p, --port *`port`*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections.

-U, --username *`username`*

Username to connect as.

-W, --password

Force password prompt.

Outputs

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.

Description

createlang is a utility for adding a new programming language to a Postgres database. createlang currently accepts several languages, `plpgsql`, `pltcl`, `pltclu`, and `plperl`.

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 more.

Notes

Use droplang to remove a language.

Usage

To install `pltcl` into the database `template1`:

```
$ createlang pltcl template1
```

droplang

Name

`droplang` Remove a programming language from a Postgres database

Synopsis

droplang [*connection-options*...] [*langname*] *dbname*
droplang [*connection-options*...] --list | -l *dbname*

Inputs

droplang accepts the following command line arguments:

langname

Specifies the name of the backend programming language to be removed. droplang will prompt for *langname* if it is not specified on the command line.

[-d, --dbname] *dbname*

Specifies from which database the language should be removed.

-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*

Specifies the hostname 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.

-p, --port *`port`*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections.

-U, --username *`username`*

Username to connect as.

-W, --password

Force password prompt.

Outputs

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.

Description

droplang is a utility for removing an existing programming language from a Postgres database. droplang currently accepts two languages, `plsql` and `pltcl`.

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.

Notes

Use createlang to add a language.

Usage

To remove `pltcl`:

```
$ droplang pltcl
```

initdb

Name

`initdb` Create a new Postgres database cluster

Synopsis

initdb --pgdata | -D *`dbdir`* [--sysid | -i *`sysid`*] [--pwprompt | -W] [--encoding | -E *`encoding`*] [-L *`directory`*] [--noclean | -n] [--debug | -d]

Description

initdb creates a new Postgres database cluster or system. A database cluster is a collection of databases that are managed by a single postmaster.

Creating a database system consists of creating the directories in which the database data will live, generating the shared catalog tables (tables that 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.

You must not execute initdb as root; it must be run by the Unix user account that will run the database server. This is because you cannot run the database server as root either, but the server needs to have access to the files initdb creates. Furthermore, during the initialization phase, when there are no users and no access controls installed, Postgres will only connect with the name of the current Unix user, so you must log in under the account that will own the server process.

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

--pgdata=*`dbdir`*
-D *`dbdir`*

> This option specifies where in the file system the database 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.

--sysid=*`sysid`*
-i *`sysid`*

> Selects the system id of the database superuser. This defaults to the effective user id of the user running initdb. It is really not important what the superuser's sysid is, but one might choose to start the numbering at some number like 1.

--pwprompt
-W

> 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.

--encoding=*`encoding`*
-E *`encoding`*

> Selects the multibyte 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 multibyte encoding feature, you must specify so at build time, at which time you also select the default for this option.

Other, less commonly used, parameters are also available:

-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.

--noclean
-n

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.

--debug
-d

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.

See also

PostgreSQL Administrator's Guide

initlocation

Name

`initlocation` Create a secondary Postgres database storage area

Synopsis

initlocation *directory*

Description

initlocation creates a new Postgres 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.

Usage

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
```

ipcclean

Name

`ipcclean` Clean up shared memory and semaphores from aborted backends

Synopsis

ipcclean

Description

ipcclean cleans up shared memory and semaphore space from aborted backends by deleting all instances owned by user `postgres`. Only the DBA should execute this program as it can cause bizarre behavior (i.e., crashes) if run during multi-user execution. This program should be executed if messages such as `semget: No space left on device` are encountered when starting up the postmaster or the backend server.

If this command is executed while 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.

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. Suggestions are welcome.

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` Starts, stops, or restarts postmaster

Synopsis

pg_ctl start [-w] [-D *datadir*] [-l *filename*] [-o *options*] [-p *path*]
pg_ctl stop [-W] [-D *datadir*] [-m s[mart] | f[ast] | i[mmediate]]
pg_ctl restart [-w] [-D *datadir*] [-m s[mart] | f[ast] | i[mmediate]] [-o *options*]
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 encapulates tasks such as redirecting log output, properly detaching from the terminal and process group, and additionally provides an option for controlled shut down.

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 on 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 will be rolled back. Immediate mode will abort without complete shutdown. This will lead to a recovery run on restart. By the default, stop mode waits for the shutdown to complete.

`restart` mode effectively executes a stop followed by a start. This allows the changing of postmaster command line options.

`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.

-w

Wait for the start or stutdown 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.

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.

Examples

Starting the postmaster

To start up postmaster:

```
$ pg_ctl start
```

An example of starting the postmaster, blocking until 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 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 then starting it again except that pg_ctl saves and reuses the command line options that were passed to the previously running instance. To restart 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.

Bugs

Waiting for complete start is not a well-defined operation and may fail if access control is set up in way that a local client cannot connect without manual interaction. It should be avoided.

See Also

postmaster, *PostgreSQL Administrator's Guide*

pg_passwd

Name

`pg_passwd` Manipulate a text password file

Synopsis

pg_passwd *filename*

Description

pg_passwd is a tool to manipulate a flat text password file for the purpose of using that file to control client authentication of the PostgreSQL server. More information about setting up this authentication mechanism can be found in the *Administrator's Guide*.

The form of a text password file is one entry per line; the fields of each entry are separated by colons. The first field is the user name, the second field is the encrypted password. Other fields are ignored (to allow password files to be shared between applications that use similar formats). The functionality of the pg_passwd utility is to enable a user to interactively add entries to such a file, to alter passwords of existing entries, and to take care of encrypting the passwords.

Supply the name of the password file as argument to the pg_passwd command. To be of use for client authentication the file needs to be located in the server's data directory, and the base name of the file needs to be specified in the `pg_hba.conf` access control file.

```
$ pg_passwd /usr/local/pgsql/data/passwords
File "/usr/local/pgsql/data/passwords" does not exist.  Create? (y/n): y
Username: guest
Password:
Re-enter password:
```

where the `Password:` and `Re-enter password:` prompts require the same password input which is not displayed on the terminal. Note that the password is limited to eight useful characters by restrictions of the standard crypt(3) library routine.

The original password file is renamed to `passwords.bk`.

To make use of this password file, put a line like the following in `pg_hba.conf`:

```
host  mydb     133.65.96.250   255.255.255.255 password passwords
```

which would allow access to database mydb from host 133.65.96.250 using the passwords listed in the `passwords` file (and only to the users listed in that file).

> **Note:** It is also useful to have entries in a password file with an empty password field. (This is different from an empty password.) These entries cannot be managed by pg_passwd, but it is always possible to edit password files manually.

See also

PostgreSQL Administrator's Guide

postgres

Name

`postgres` Run a PostgreSQL single-user backend

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] [-L] [-N] [-o *file-name*] [-O] [-P] [-s | -t pa | pl | ex] [-S *sort-mem*] [-W *seconds*] *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] [-L] [-o *file-name*] [-O] [-p *database*] [-P] [-s | -t pa | pl | ex] [-S *sort-mem*] [-v *protocol-version*] [-W *seconds*]

Description

The `postgres` executable is the actual PostgreSQL server process that processes queries. It is normally not called directly; instead a postmaster multi-user 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 mode. The primary use for this mode is for bootstrapping by initdb.

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 inter-process communication and locking will happen.

When running a stand-alone backend the session user name will automatically be set to the current effective Unix user name. If that user does not exist the server will not start.

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`, and `-F` have the same meaning as with the postmaster.

-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 *`file-name`*

Sends all debugging and error output to *`OutputFile`*. If the backend is running under the postmaster, error messages are still sent to the frontend process as well as to *`OutputFile`*, but debugging output is sent to the controlling tty of the postmaster (since only one file descriptor can be sent to an actual file).

-P

Ignore system indexes to scan/update 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.

-L

Turns off the locking system.

-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.

See also

initdb, ipcclean, postmaster

postmaster

Name

`postmaster` PostgreSQL multi-user 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] [-n | -s]

Description

postmaster is the PostgreSQL multi-user 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.

On some systems it is also possible to equivalently use GNU-style long options in the form `--name=value`.

-d *`debug-level`*

Sets the debug level. The higher this value is set, the more debugging output is written to the server log. The default is 0, which means no debugging. Values up to 4 make sense.

-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. Read the detailed documentation before using this!

-h *`hostname`*

Specifies the TCP/IP hostname 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.

-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 1024 if your system will support that many processes. (Note that `-B` is required to be at least twice `-N`.)

-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.

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.

Outputs

`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 Postgres' shared memory consumption, and/or by reducing -N to reduce Postgres' 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 lengthy 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.

Usage

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
```